The Python Library Reference

發🄵 *3.14.0rc3*

Guido van Rossum and the Python development team

10 月 01, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 簡介 **3**
- 1.1 可用性之標🄴 . . . 3
  - 1.1.1 WebAssembly 平台 . . . 4
  - 1.1.2 行動平台 . . . 4

2 🄴建函式 **7**

3 🄴建常數 **35**
- 3.1 由 site module（模組）所添增的常數 . . . 36

4 🄴建型🄴 **37**
- 4.1 真值檢測 . . . 37
- 4.2 Boolean（布林）運算 --- and, or, not . . . 37
- 4.3 比較運算 . . . 38
- 4.4 數值型🄴 --- int、float、complex . . . 38
  - 4.4.1 整數型🄴的位元運算 . . . 40
  - 4.4.2 整數型🄴的附加方法 . . . 40
  - 4.4.3 浮點數的附加方法 . . . 42
  - 4.4.4 🄴數的附加方法 . . . 43
  - 4.4.5 數值型🄴的雜🄴 . . . 44
- 4.5 Boolean 型🄴 - bool . . . 45
- 4.6 🄴代器型🄴 . . . 45
  - 4.6.1 Generator Types . . . 46
- 4.7 Sequence Types --- list, tuple, range . . . 46
  - 4.7.1 Common Sequence Operations . . . 46
  - 4.7.2 不可變序列型🄴 . . . 48
  - 4.7.3 可變序列型🄴 . . . 48
  - 4.7.4 List（串列） . . . 49
  - 4.7.5 Tuples . . . 50
  - 4.7.6 Ranges . . . 51
- 4.8 Text and Binary Sequence Type Methods Summary . . . 52
- 4.9 Text Sequence Type --- str . . . 53
  - 4.9.1 字串方法 . . . 54
  - 4.9.2 Formatted String Literals (f-strings) . . . 63
  - 4.9.3 printf-style String Formatting . . . 64
- 4.10 Binary Sequence Types --- bytes, bytearray, memoryview . . . 66
  - 4.10.1 Bytes Objects . . . 67
  - 4.10.2 Bytearray 物件 . . . 68
  - 4.10.3 Bytes 和 Bytearray 的操作 . . . 69
  - 4.10.4 printf-style Bytes Formatting . . . 80
  - 4.10.5 Memory Views . . . 82

4.11 Set Types --- set, frozenset . . . 89
4.12 Mapping Types --- dict . . . 91
4.12.1 字典視圖物件 . . . 95
4.13 情境管理器型F . . . 96
4.14 型FF釋的型F --- 泛型F名 (Generic Alias)、聯合 (Union) . . . 97
4.14.1 泛型F名型F . . . 97
4.14.2 聯合型F (Union Type) . . . 101
4.15 其他F建型F . . . 102
4.15.1 模組 . . . 103
4.15.2 類F與類F實例 . . . 103
4.15.3 函式 . . . 103
4.15.4 方法 . . . 103
4.15.5 程式碼物件 . . . 104
4.15.6 Type Objects . . . 104
4.15.7 Null 物件 . . . 104
4.15.8 Ellipsis 物件 . . . 104
4.15.9 NotImplemented 物件 . . . 104
4.15.10 F部物件 . . . 104
4.16 特殊屬性 . . . 105
4.17 Integer string conversion length limitation . . . 105
4.17.1 受影響的 API . . . 106
4.17.2 設定限制 . . . 106
4.17.3 建議的配置 . . . 107

5 F建的例外 **109**
5.1 例外的情境 . . . 109
5.2 繼承自F建的例外 . . . 110
5.3 基底類F (base classes) . . . 110
5.4 實體例外 . . . 111
5.4.1 作業系統例外 . . . 116
5.5 警告 . . . 117
5.6 例外群組 . . . 118
5.7 例外階層 . . . 120

6 文本處理 (Text Processing) 服務 **123**
6.1 string --- 常見的字串操作 . . . 123
6.1.1 字串常數 . . . 123
6.1.2 自訂字串格式 . . . 124
6.1.3 格式化文字語法 . . . 125
6.1.4 模板字串 ($-strings) . . . 132
6.1.5 輔助函式 . . . 134
6.2 string.templatelib --- 對模板字串字面值的支援 . . . 134
6.2.1 模板字串 . . . 134
6.2.2 Types . . . 135
6.2.3 Helper functions . . . 139
6.3 re --- 正規表示式 (regular expression) 操作 . . . 139
6.3.1 正規表示式語法 . . . 139
6.3.2 模組F容 . . . 146
6.3.3 Regular Expression Objects . . . 152
6.3.4 Match Objects . . . 153
6.3.5 Regular Expression Examples . . . 156
6.4 difflib --- 計算差F的輔助工具 . . . 161
6.4.1 SequenceMatcher 物件 . . . 165
6.4.2 SequenceMatcher 範例 . . . 168
6.4.3 Differ Objects . . . 168
6.4.4 Differ Example . . . 169
6.4.5 difflib 的命令列介面 . . . 170
6.4.6 ndiff 範例: . . . 171

6.5 textwrap --- 文字包裝與填充 . . . 173
6.6 unicodedata --- Unicode 資料庫 . . . 176
6.7 stringprep --- 網際網路字串的准備 . . . 179
6.8 readline --- GNU readline 介面 . . . 180
6.8.1 Init file . . . 181
6.8.2 Line buffer . . . 181
6.8.3 History file . . . 181
6.8.4 History list . . . 182
6.8.5 Startup hooks . . . 182
6.8.6 Completion . . . 182
6.8.7 範例 . . . 183
6.9 rlcompleter --- GNU readline 的補全函式 . . . 184

7 二進位資料服務 187
7.1 struct --- 將位元組直譯🄵打包起來的二進位資料 . . . 187
7.1.1 函式與例外 . . . 187
7.1.2 格式字串 . . . 188
7.1.3 應用 . . . 192
7.1.4 類🄵 . . . 193
7.2 codecs --- 編解碼器🄵🄵表和基底類🄵 . . . 194
7.2.1 Codec Base Classes . . . 197
7.2.2 編碼和 Unicode . . . 203
7.2.3 標准編碼 . . . 204
7.2.4 Python Specific Encodings . . . 207
7.2.5 encodings --- Encodings package . . . 209
7.2.6 encodings.idna --- Internationalized Domain Names in Applications . . . 210
7.2.7 encodings.mbcs --- Windows ANSI 碼頁 . . . 211
7.2.8 encodings.utf_8_sig --- UTF-8 codec with BOM signature . . . 211

8 資料型🄵 213
8.1 datetime --- 日期與時間的基本型🄵 . . . 213
8.1.1 Aware and Naive Objects . . . 214
8.1.2 常數 . . . 214
8.1.3 Available Types . . . 214
8.1.4 timedelta 物件 . . . 215
8.1.5 date 物件 . . . 219
8.1.6 datetime 物件 . . . 224
8.1.7 time 物件 . . . 236
8.1.8 tzinfo 物件 . . . 239
8.1.9 timezone 物件 . . . 246
8.1.10 strftime() 與 strptime() 的行🄵 . . . 246
8.2 zoneinfo --- IANA 時區支援 . . . 251
8.2.1 使用 ZoneInfo . . . 251
8.2.2 資料來源 . . . 252
8.2.3 ZoneInfo 類🄵 . . . 253
8.2.4 函式 . . . 255
8.2.5 全域變數 . . . 255
8.2.6 例外與警告 . . . 255
8.3 calendar --- 日🄵相關函式 . . . 255
8.3.1 命令列用法 . . . 262
8.4 collections --- 容器資料型態 . . . 263
8.4.1 ChainMap 物件 . . . 264
8.4.2 Counter 物件 . . . 266
8.4.3 deque 物件 . . . 269
8.4.4 defaultdict 物件 . . . 272
8.4.5 namedtuple() 擁有具名欄位之 tuple 的工廠函式 . . . 273
8.4.6 OrderedDict 物件 . . . 276
8.4.7 UserDict 物件 . . . 279

8.4.8 UserList 物件 . . . 279
8.4.9 UserString 物件 . . . 279
8.5 collections.abc --- 容器的抽象基底類F . . . 280
8.5.1 Collections Abstract Base Classes . . . 281
8.5.2 Collections Abstract Base Classes -- Detailed Descriptions . . . 283
8.5.3 Examples and Recipes . . . 285
8.6 heapq --- 堆積F列 (heap queue) 演算法 . . . 286
8.6.1 基礎範例 . . . 288
8.6.2 優先F列實作細節 . . . 288
8.6.3 原理 . . . 289
8.7 bisect --- 陣列二分演算法 (Array bisection algorithm) . . . 290
8.7.1 效能考量 . . . 291
8.7.2 搜尋一個已排序的 list . . . 292
8.7.3 範例 . . . 292
8.8 array --- 高效率的數值型陣列 . . . 293
8.9 weakref --- 弱參照 . . . 296
8.9.1 弱參照物件 . . . 300
8.9.2 範例 . . . 301
8.9.3 最終化器物件 . . . 301
8.9.4 最終化器與 __del__() 方法的比較 . . . 302
8.10 types --- 動態型F建立與F建型F名稱 . . . 303
8.10.1 Dynamic Type Creation . . . 303
8.10.2 Standard Interpreter Types . . . 305
8.10.3 Additional Utility Classes and Functions . . . 308
8.10.4 Coroutine Utility Functions . . . 309
8.11 copy --- 淺層 (shallow) 和深層 (deep) FF操作 . . . 309
8.12 pprint --- 資料美化列印器 . . . 311
8.12.1 函式 . . . 311
8.12.2 PrettyPrinter 物件 . . . 312
8.12.3 範例 . . . 313
8.13 reprlib --- repr() 的替代實作 . . . 316
8.13.1 Repr 物件 . . . 317
8.13.2 Subclassing Repr Objects . . . 319
8.14 enum --- 對列舉的支援 . . . 319
8.14.1 模組F容 . . . 320
8.14.2 資料型F . . . 321
8.14.3 通用項目與裝飾器 . . . 332
8.14.4 備F . . . 334
8.15 graphlib —-- 使用類圖 (graph-like) 結構進行操作的功能 . . . 334
8.15.1 例外 . . . 336

9 數值與數學模組 337
9.1 numbers --- 數值的抽象基底類F . . . 337
9.1.1 數值的階層 . . . 337
9.1.2 給型F實作者的F記 . . . 338
9.2 math --- 數學函式 . . . 340
9.2.1 數論函式 . . . 341
9.2.2 Floating point arithmetic . . . 342
9.2.3 Floating point manipulation functions . . . 343
9.2.4 Power, exponential and logarithmic functions . . . 345
9.2.5 Summation and product functions . . . 346
9.2.6 Angular conversion . . . 346
9.2.7 Trigonometric functions . . . 347
9.2.8 Hyperbolic functions . . . 347
9.2.9 特殊函式 . . . 347
9.2.10 常數 . . . 348
9.3 cmath --- F數的數學函式 . . . 349
9.3.1 轉F到極座標和從極座標做轉F . . . 350

9.3.2 Ⓕ函數和對數函數 . . . 351
9.3.3 三角函數 . . . 351
9.3.4 雙曲函數 . . . 351
9.3.5 分類函式 . . . 352
9.3.6 常數 . . . 352
9.4 decimal --- 十進位固定點和浮點運算 . . . 353
9.4.1 Quick-start tutorial . . . 354
9.4.2 Decimal objects . . . 357
9.4.3 Context objects . . . 365
9.4.4 常數 . . . 371
9.4.5 Rounding modes . . . 371
9.4.6 Signals . . . 372
9.4.7 Floating-point notes . . . 373
9.4.8 Working with threads . . . 375
9.4.9 Recipes . . . 375
9.4.10 Decimal FAQ . . . 378
9.5 fractions --- 有理數 . . . 381
9.6 random --- Ⓕ生Ⓕ隨機數 . . . 384
9.6.1 簿記函式 (bookkeeping functions) . . . 385
9.6.2 回傳位元組的函式 . . . 385
9.6.3 回傳整數的函式 . . . 386
9.6.4 回傳序列的函式 . . . 386
9.6.5 離散分布 . . . 387
9.6.6 實數分布 . . . 387
9.6.7 替代Ⓕ生器 . . . 388
9.6.8 關於 Reproducibility（復現性）的注意事項 . . . 389
9.6.9 範例 . . . 389
9.6.10 使用方案 . . . 391
9.6.11 命令列用法 . . . 392
9.6.12 命令列範例 . . . 393
9.7 statistics --- 數學統計函式 . . . 393
9.7.1 平均值與中央位置量數 . . . 394
9.7.2 離度 (spread) 的測量 . . . 394
9.7.3 兩個輸入之間的關Ⓕ統計 . . . 395
9.7.4 函式細節 . . . 395
9.7.5 例外 . . . 403
9.7.6 NormalDist 物件 . . . 404
9.7.7 範例與錦囊妙計 . . . 405

10 函式程式設計模組 **409**
10.1 itertools --- 建立Ⓕ生高效率Ⓕ圈之Ⓕ代器的函式 . . . 409
10.1.1 Itertool 函式 . . . 411
10.1.2 Itertools 應用技巧 . . . 420
10.2 functools --- 可呼叫物件上的高階函式與操作 . . . 425
10.2.1 partial 物件 . . . 435
10.3 operator --- 標准運算子替代函式 . . . 435
10.3.1 運算子與函式間的對映 . . . 440
10.3.2 原地 (in-place) 運算子 . . . 440

11 檔案與目Ⓕ存取 **443**
11.1 pathlib --- 物件導向檔案系統路徑 . . . 443
11.1.1 基本用法 . . . 444
11.1.2 例外 . . . 445
11.1.3 純路徑 . . . 445
11.1.4 實體路徑 . . . 454
11.1.5 模式語言 (pattern language) . . . 466
11.1.6 與 glob 模組的比較 . . . 466
11.1.7 與 os 和 os.path 模組的比較 . . . 467

11.1.8 Protocols . . . 468
11.2 os.path --- 常見的路徑名操作 . . . 468
11.3 stat --- 直譯 stat() 的結果 . . . 475
11.4 filecmp --- 檔案與目F比較 . . . 481
11.4.1 The dircmp class . . . 481
11.5 tempfile --- 生成臨時檔案和目F . . . 483
11.5.1 範例 . . . 487
11.5.2 已F用的函式和變數 . . . 487
11.6 glob --- Unix 風格的路徑名稱模式擴展 . . . 488
11.6.1 範例 . . . 490
11.7 fnmatch --- Unix 檔案名稱模式比對 . . . 490
11.8 linecache --- 隨機存取文字列 . . . 492
11.9 shutil —高階檔案操作 . . . 492
11.9.1 目F和檔案操作 . . . 493
11.9.2 Archiving operations . . . 499
11.9.3 Querying the size of the output terminal . . . 502

12 資料持久性 (Data Persistence) 505
12.1 pickle --- Python 物件序列化 . . . 505
12.1.1 和其他 Python 模組的關F . . . 505
12.1.2 資料串流格式 . . . 506
12.1.3 模組介面 . . . 507
12.1.4 哪些物件能或不能被封裝、拆封? . . . 510
12.1.5 Pickling 類F實例 . . . 511
12.1.6 針對型F、函式或特定物件定F縮F函式 . . . 516
12.1.7 帶外（Out-of-band）資料緩衝區 . . . 517
12.1.8 限制全域物件 . . . 518
12.1.9 效能 . . . 519
12.1.10 範例 . . . 519
12.1.11 命令列介面 . . . 520
12.2 copyreg --- FF pickle 支援函式 . . . 520
12.2.1 範例 . . . 521
12.3 shelve --- Python object persistence . . . 521
12.3.1 Restrictions . . . 522
12.3.2 範例 . . . 523
12.4 marshal --- F部 Python 物件序列化 . . . 524
12.5 dbm --- Unix "databases" 的介面 . . . 525
12.5.1 dbm.sqlite3 --- dbm 的 SQLite 後端 . . . 527
12.5.2 dbm.gnu --- GNU 資料庫管理器 . . . 528
12.5.3 dbm.ndbm --- 新資料庫管理器 . . . 529
12.5.4 dbm.dumb --- 可F式 DBM 實作 . . . 530
12.6 sqlite3 --- SQLite 資料庫的 DB-API 2.0 介面 . . . 531
12.6.1 教學 . . . 532
12.6.2 Reference . . . 534
12.6.3 How-to guides . . . 554
12.6.4 解釋 . . . 561

13 資料壓縮與保存 563
13.1 compression 套件 . . . 563
13.2 compression.zstd --- 與 Zstandard 格式相容的壓縮 . . . 563
13.2.1 例外 . . . 564
13.2.2 Reading and writing compressed files . . . 564
13.2.3 Compressing and decompressing data in memory . . . 565
13.2.4 Zstandard 字典 . . . 567
13.2.5 Advanced parameter control . . . 569
13.2.6 Miscellaneous . . . 573
13.2.7 範例 . . . 573
13.3 zlib --- 相容於 **gzip** 的壓縮 . . . 574

13.4 gzip --- **gzip** 檔案的支援 . . . 577
13.4.1 用法範例 . . . 580
13.4.2 命令列介面 . . . 580
13.5 bz2 --- **bzip2** 壓縮的支援 . . . 581
13.5.1 (De)compression of files . . . 581
13.5.2 Incremental (de)compression . . . 583
13.5.3 One-shot (de)compression . . . 584
13.5.4 用法範例 . . . 584
13.6 lzma --- 使用 LZMA 演算法進行壓縮 . . . 585
13.6.1 Reading and writing compressed files . . . 586
13.6.2 Compressing and decompressing data in memory . . . 587
13.6.3 Miscellaneous . . . 589
13.6.4 Specifying custom filter chains . . . 589
13.6.5 範例 . . . 590
13.7 zipfile --- 處理 ZIP 封存檔案 . . . 591
13.7.1 ZipFile 物件 . . . 593
13.7.2 Path 物件 . . . 597
13.7.3 PyZipFile 物件 . . . 598
13.7.4 ZipInfo 物件 . . . 599
13.7.5 命令列介面 . . . 601
13.7.6 解壓縮的陷阱 . . . 601
13.8 tarfile --- 讀取與寫入 tar 封存檔案 . . . 602
13.8.1 TarFile 物件 . . . 606
13.8.2 TarInfo 物件 . . . 610
13.8.3 Extraction filters . . . 612
13.8.4 命令列介面 . . . 615
13.8.5 範例 . . . 616
13.8.6 支援的 tar 格式 . . . 617
13.8.7 Unicode issues . . . 618

14 檔案格式 **619**
14.1 csv --- CSV 檔案讀取及寫入 . . . 619
14.1.1 模組🄵容 . . . 619
14.1.2 Dialect 與格式參數 . . . 623
14.1.3 讀取器物件 . . . 624
14.1.4 寫入器物件 . . . 624
14.1.5 範例 . . . 625
14.2 configparser --- 設定檔剖析器 . . . 626
14.2.1 Quick Start . . . 627
14.2.2 Supported Datatypes . . . 628
14.2.3 Fallback Values . . . 629
14.2.4 支援的 INI 檔案結構 . . . 629
14.2.5 Unnamed Sections . . . 630
14.2.6 Interpolation of values . . . 631
14.2.7 Mapping Protocol Access . . . 632
14.2.8 Customizing Parser Behaviour . . . 632
14.2.9 Legacy API Examples . . . 637
14.2.10 ConfigParser 物件 . . . 638
14.2.11 RawConfigParser 物件 . . . 642
14.2.12 例外 . . . 643
14.3 tomllib --- 剖析 TOML 檔案 . . . 644
14.3.1 範例 . . . 645
14.3.2 轉🄵表 . . . 646
14.4 netrc --- netrc 檔案處理 . . . 646
14.4.1 netrc 物件 . . . 647
14.5 plistlib --- 🄵生和剖析 Apple .plist 檔案 . . . 647
14.5.1 範例 . . . 649

15 加密服務 **651**
15.1 hashlib --- 安全雜F與訊息摘要 . . . 651
15.1.1 雜F演算法 . . . 651
15.1.2 用法 . . . 652
15.1.3 建構函式 . . . 652
15.1.4 屬性 . . . 653
15.1.5 雜F物件 . . . 653
15.1.6 SHAKE 可變長度摘要 . . . 654
15.1.7 檔案雜F . . . 654
15.1.8 密鑰的生成 . . . 655
15.1.9 BLAKE2 . . . 655
15.2 hmac --- 基於金鑰雜F的訊息驗證 . . . 662
15.3 secrets --- F生用於管理機密的安全亂數 . . . 663
15.3.1 亂數 . . . 664
15.3.2 F生權杖（token） . . . 664
15.3.3 其他函式 . . . 665
15.3.4 應用技巧和典範實務（best practices） . . . 665

16 通用作業系統服務 **667**
16.1 os --- 各種作業系統介面 . . . 667
16.1.1 File Names, Command Line Arguments, and Environment Variables . . . 668
16.1.2 Python UTF-8 模式 . . . 668
16.1.3 行程參數 . . . 669
16.1.4 File Object Creation . . . 677
16.1.5 File Descriptor Operations . . . 677
16.1.6 Files and Directories . . . 690
16.1.7 行程管理 . . . 716
16.1.8 排程器介面 . . . 729
16.1.9 Miscellaneous System Information . . . 731
16.1.10 Random numbers . . . 733
16.2 io —處理資料串流的核心工具 . . . 734
16.2.1 總覽 . . . 734
16.2.2 文字編碼 . . . 735
16.2.3 高階模組介面 . . . 735
16.2.4 類F階層 . . . 736
16.2.5 Static Typing . . . 746
16.2.6 Performance . . . 747
16.3 time --- 時間存取與轉F . . . 748
16.3.1 函式 . . . 748
16.3.2 時鐘 ID 常數 . . . 757
16.3.3 時區常數 . . . 759
16.4 logging --- Python 的日F記F工具 . . . 759
16.4.1 Logger 物件 . . . 761
16.4.2 Logging Levels . . . 765
16.4.3 Handler Objects . . . 766
16.4.4 Formatter Objects . . . 768
16.4.5 Filter Objects . . . 769
16.4.6 LogRecord 物件 . . . 770
16.4.7 LogRecord 屬性 . . . 771
16.4.8 LoggerAdapter 物件 . . . 773
16.4.9 執行緒安全 . . . 773
16.4.10 模組層級函式 . . . 773
16.4.11 模組層級屬性 . . . 777
16.4.12 Integration with the warnings module . . . 778
16.5 logging.config --- 日F記F配置 . . . 778
16.5.1 Configuration functions . . . 779
16.5.2 Security considerations . . . 781
16.5.3 Configuration dictionary schema . . . 781

16.5.4 Configuration file format . . . 787
16.6 logging.handlers --- 日🄵紀🄵處理器 . . . 790
16.6.1 StreamHandler . . . 790
16.6.2 FileHandler . . . 791
16.6.3 NullHandler . . . 791
16.6.4 WatchedFileHandler . . . 792
16.6.5 BaseRotatingHandler . . . 792
16.6.6 RotatingFileHandler . . . 794
16.6.7 TimedRotatingFileHandler . . . 794
16.6.8 SocketHandler . . . 795
16.6.9 DatagramHandler . . . 796
16.6.10 SysLogHandler . . . 797
16.6.11 NTEventLogHandler . . . 799
16.6.12 SMTPHandler . . . 800
16.6.13 MemoryHandler . . . 800
16.6.14 HTTPHandler . . . 801
16.6.15 QueueHandler . . . 802
16.6.16 QueueListener . . . 803
16.7 platform --- 對底層平臺識🄵資料的存取 . . . 804
16.7.1 跨平台 . . . 804
16.7.2 Java 平台 . . . 806
16.7.3 Windows 平台 . . . 806
16.7.4 macOS 平台 . . . 806
16.7.5 iOS 平台 . . . 807
16.7.6 Unix 平台 . . . 807
16.7.7 Linux 平台 . . . 807
16.7.8 Android 平台 . . . 808
16.7.9 命令列用法 . . . 808
16.8 errno --- 標准 errno 系統符號 . . . 808
16.9 ctypes --- 用於 Python 的外部函式庫 . . . 817
16.9.1 ctypes 教學 . . . 817
16.9.2 ctypes reference . . . 834

17 命令列介面函式庫 **855**
17.1 argparse --- 命令列選項、引數和子命令的剖析器 . . . 855
17.1.1 ArgumentParser 物件 . . . 856
17.1.2 add_argument() 方法 . . . 864
17.1.3 parse_args() 方法 . . . 874
17.1.4 Other utilities . . . 877
17.1.5 例外 . . . 885
17.2 optparse --- 命令列選項剖析器 . . . 898
17.2.1 選擇一個命令列參數剖析函式庫 . . . 898
17.2.2 Introduction . . . 900
17.2.3 背景 . . . 900
17.2.4 教學 . . . 902
17.2.5 Reference Guide . . . 909
17.2.6 Option Callbacks . . . 918
17.2.7 Extending optparse . . . 922
17.2.8 例外 . . . 925
17.3 getpass --- 可🄵式密碼輸入工具 . . . 925
17.4 fileinput --- 逐列🄵代多個輸入串流 . . . 926
17.5 curses --- 字元儲存格顯示的終端處理 . . . 928
17.5.1 函式 . . . 929
17.5.2 Window Objects . . . 936
17.5.3 Constants . . . 942
17.6 curses.textpad --- Text input widget for curses programs . . . 953
17.6.1 Textbox objects . . . 953
17.7 curses.ascii --- ASCII 字元的工具程式 . . . 954

17.8 curses.panel --- curses 的面板堆🅵擴充 . . . 958
17.8.1 函式 . . . 958
17.8.2 Panel Objects . . . 958
17.9 cmd --- 以列🅵導向的命令直譯器支援 . . . 959
17.9.1 Cmd 物件 . . . 960
17.9.2 Cmd 範例 . . . 961

18 🅵行執行 (Concurrent Execution) 965
18.1 threading --- 基於執行緒的平行性 . . . 965
18.1.1 Introduction . . . 965
18.1.2 GIL and performance considerations . . . 966
18.1.3 Reference . . . 966
18.1.4 Using locks, conditions, and semaphores in the with statement . . . 982
18.2 multiprocessing --- 以行程🅵基礎的平行性 . . . 982
18.2.1 簡介 . . . 982
18.2.2 Reference . . . 989
18.2.3 Programming guidelines . . . 1017
18.2.4 範例 . . . 1020
18.3 multiprocessing.shared_memory --- 對於共享記憶體的跨行程直接存取 . . . 1026
18.4 concurrent 套件 . . . 1031
18.5 concurrent.futures --- 🅵動平行任務 . . . 1031
18.5.1 Executor 物件 . . . 1032
18.5.2 ThreadPoolExecutor . . . 1033
18.5.3 InterpreterPoolExecutor . . . 1034
18.5.4 ProcessPoolExecutor . . . 1035
18.5.5 Future 物件 . . . 1037
18.5.6 模組函式 . . . 1038
18.5.7 例外類🅵 . . . 1039
18.6 concurrent.interpreters --- 在同一個行程中使用多個直譯器 . . . 1039
18.6.1 Key details . . . 1040
18.6.2 Introduction . . . 1040
18.6.3 Reference . . . 1042
18.6.4 基本用法 . . . 1043
18.7 subprocess --- 子行程管理 . . . 1044
18.7.1 Using the subprocess Module . . . 1044
18.7.2 安全性注意事項 . . . 1053
18.7.3 Popen Objects . . . 1053
18.7.4 Windows Popen Helpers . . . 1055
18.7.5 Older high-level API . . . 1057
18.7.6 Replacing Older Functions with the subprocess Module . . . 1059
18.7.7 Legacy Shell Invocation Functions . . . 1061
18.7.8 🅵解 . . . 1062
18.8 sched --- 事件排程器 . . . 1063
18.8.1 排程器物件 . . . 1063
18.9 queue --- 同步🅵列 (synchronized queue) 類🅵 . . . 1064
18.9.1 🅵列物件 . . . 1065
18.9.2 SimpleQueue 物件 . . . 1067
18.10 contextvars --- 情境變數 . . . 1067
18.10.1 情境變數 . . . 1068
18.10.2 手動情境管理 . . . 1069
18.10.3 對 asyncio 的支援 . . . 1071
18.11 _thread --- 低階執行緒 API . . . 1071

19 網路與行程間通訊 1075
19.1 asyncio --- 非同步 I/O . . . 1075
19.1.1 Runners (執行器) . . . 1076
19.1.2 協程與任務 . . . 1078
19.1.3 串流 . . . 1098

19.1.4 同步化原始物件 (Synchronization Primitives) . . . 1105
19.1.5 子行程 . . . 1111
19.1.6 🄵列 (Queues) . . . 1115
19.1.7 例外 . . . 1118
19.1.8 Call Graph Introspection . . . 1119
19.1.9 事件🄵圈 . . . 1120
19.1.10 Futures . . . 1143
19.1.11 傳輸與協定 . . . 1147
19.1.12 Policies . . . 1160
19.1.13 平臺支援 . . . 1161
19.1.14 擴充 . . . 1162
19.1.15 高階 API 索引 . . . 1164
19.1.16 低階 API 索引 . . . 1166
19.1.17 使用 asyncio 開發 . . . 1170
19.2 socket —低階網路介面 . . . 1173
19.2.1 Socket 系列家族 . . . 1174
19.2.2 模組🄵容 . . . 1177
19.2.3 Socket 物件 . . . 1191
19.2.4 Notes on socket timeouts . . . 1198
19.2.5 範例 . . . 1199
19.3 ssl --- socket 物件的 TLS/SSL 包裝器 . . . 1202
19.3.1 函式、常數與例外 . . . 1203
19.3.2 SSL Sockets . . . 1214
19.3.3 SSL Contexts . . . 1218
19.3.4 Certificates . . . 1227
19.3.5 範例 . . . 1229
19.3.6 Notes on non-blocking sockets . . . 1232
19.3.7 記憶體 BIO 支援 . . . 1232
19.3.8 SSL session . . . 1234
19.3.9 Security considerations . . . 1234
19.3.10 TLS 1.3 . . . 1236
19.4 select --- 等待 I/O 完成 . . . 1237
19.4.1 /dev/poll Polling Objects . . . 1239
19.4.2 Edge and Level Trigger Polling (epoll) Objects . . . 1240
19.4.3 Polling Objects . . . 1241
19.4.4 Kqueue Objects . . . 1241
19.4.5 Kevent Objects . . . 1242
19.5 selectors --- 高階 I/O 多工 . . . 1243
19.5.1 簡介 . . . 1244
19.5.2 Classes . . . 1244
19.5.3 範例 . . . 1246
19.6 signal --- 設定非同步事件的處理函式 . . . 1247
19.6.1 一般規則 . . . 1247
19.6.2 模組🄵容 . . . 1248
19.6.3 範例 . . . 1254
19.6.4 關於 SIGPIPE 的🄵明 . . . 1254
19.6.5 訊號處理程式與例外的🄵明 . . . 1255
19.7 mmap --- 記憶體對映檔案的支援 . . . 1256
19.7.1 MADV_* 常數 . . . 1260
19.7.2 MAP_* 常數 . . . 1260

20 網際網路資料處理 1263
20.1 email --- 郵件和 MIME 處理套件 . . . 1263
20.1.1 email.message: 表示電子郵件訊息 . . . 1264
20.1.2 email.parser: 剖析電子郵件訊息 . . . 1272
20.1.3 email.generator: 🄵生 MIME 文件 . . . 1275
20.1.4 email.policy: Policy Objects . . . 1278
20.1.5 email.errors: 例外和缺陷類🄵 . . . 1285

20.1.6 email.headerregistry: 自訂標頭物件 . . . 1286
20.1.7 email.contentmanager: 管理 MIME 🄵容 . . . 1292
20.1.8 email: 範例 . . . 1294
20.1.9 email.message.Message: Representing an email message using the compat32 API . . . 1300
20.1.10 email.mime: 從頭開始建立電子郵件和 MIME 物件 . . . 1308
20.1.11 email.header: 國際化標頭 . . . 1311
20.1.12 email.charset: 字元集合的表示 . . . 1313
20.1.13 email.encoders: 編碼器 . . . 1315
20.1.14 email.utils: 雜項工具 . . . 1316
20.1.15 email.iterators: 🄵代器 . . . 1318
20.2 json --- JSON 編碼器與解碼器 . . . 1319
20.2.1 基本用法 . . . 1322
20.2.2 編碼器與解碼器 . . . 1324
20.2.3 例外 . . . 1326
20.2.4 合規性與互通性（Interoperability） . . . 1326
20.2.5 命令列介面 . . . 1328
20.3 mailbox --- 以各種格式操作郵件信箱 . . . 1329
20.3.1 Mailbox 物件 . . . 1329
20.3.2 Message 物件 . . . 1339
20.3.3 例外 . . . 1346
20.3.4 範例 . . . 1347
20.4 mimetypes --- 將檔案名稱對映到 MIME 類型 . . . 1348
20.4.1 MimeTypes 物件 . . . 1350
20.4.2 Command-line usage . . . 1351
20.4.3 Command-line example . . . 1352
20.5 base64 --- Base16、Base32、Base64、Base85 資料編碼 . . . 1352
20.5.1 RFC 4648 編碼 . . . 1353
20.5.2 Base85 編碼 . . . 1354
20.5.3 舊版介面 . . . 1355
20.5.4 安全性注意事項 . . . 1356
20.6 binascii --- 在二進位制和 ASCII 之間轉🄵 . . . 1356
20.7 quopri --- 編碼和解碼 MIME 可列印字元資料 . . . 1358

21 結構化標記處理工具 1359
21.1 html --- 超文本標記語言 (HTML) 支援 . . . 1359
21.2 html.parser --- 簡單的 HTML 和 XHTML 剖析器 . . . 1360
21.2.1 HTML 剖析器應用程式範例 . . . 1360
21.2.2 HTMLParser 方法 . . . 1361
21.2.3 範例 . . . 1362
21.3 html.entities --- HTML 一般實體的定義 . . . 1364
21.4 XML 處理模組 . . . 1364
21.4.1 XML 安全性 . . . 1365
21.5 xml.etree.cElementTree --- ElementTree XML API . . . 1365
21.5.1 教學 . . . 1366
21.5.2 XPath 支援 . . . 1370
21.5.3 Reference . . . 1372
21.5.4 XInclude support . . . 1375
21.5.5 Reference . . . 1376
21.6 xml.dom --- Document 物件模型 API . . . 1384
21.6.1 模組🄵容 . . . 1385
21.6.2 Objects in the DOM . . . 1386
21.6.3 Conformance . . . 1393
21.7 xml.dom.minidom --- 最小的 DOM 實作 . . . 1394
21.7.1 DOM 物件 . . . 1395
21.7.2 DOM 範例 . . . 1396
21.7.3 minidom 和 DOM 標準 . . . 1397
21.8 xml.dom.pulldom --- 支援建置部分 DOM 樹 . . . 1398
21.8.1 DOMEventStream 物件 . . . 1399

21.9 xml.sax --- SAX2 剖析器支援 . . . 1400
21.9.1 SAXException 物件 . . . 1401
21.10 xml.sax.handler --- SAX 處理函式的基本類F . . . 1401
21.10.1 ContentHandler 物件 . . . 1403
21.10.2 DTDHandler 物件 . . . 1405
21.10.3 EntityResolver 物件 . . . 1405
21.10.4 ErrorHandler 物件 . . . 1406
21.10.5 LexicalHandler 物件 . . . 1406
21.11 xml.sax.saxutils --- SAX 工具程式 . . . 1406
21.12 xml.sax.xmlreader --- XML 剖析器的介面 . . . 1407
21.12.1 XMLReader 物件 . . . 1408
21.12.2 IncrementalParser 物件 . . . 1409
21.12.3 Locator Objects . . . 1410
21.12.4 InputSource 物件 . . . 1410
21.12.5 The Attributes Interface . . . 1411
21.12.6 The AttributesNS Interface . . . 1411
21.13 xml.parsers.expat --- 使用 Expat 進行快速 XML 剖析 . . . 1411
21.13.1 XMLParser 物件 . . . 1412
21.13.2 ExpatError 例外 . . . 1416
21.13.3 範例 . . . 1417
21.13.4 Content Model Descriptions . . . 1418
21.13.5 Expat 錯誤常數 . . . 1418

22 網路協定 (Internet protocols) 及支援 1423
22.1 webbrowser --- 方便的網頁F覽器控制器 . . . 1423
22.1.1 F覽器控制器物件 . . . 1425
22.2 wsgiref --- WSGI 工具與參考實作 . . . 1426
22.2.1 wsgiref.util -- WSGI 環境工具 . . . 1426
22.2.2 wsgiref.headers -- WSGI 回應標頭工具 . . . 1428
22.2.3 wsgiref.simple_server -- 一個簡單的 WSGI HTTP 伺服器 . . . 1428
22.2.4 wsgiref.validate --- WSGI 符合性檢查 . . . 1429
22.2.5 wsgiref.handlers -- 伺服器 / 閘道基本類F . . . 1430
22.2.6 wsgiref.types -- 用於F態型F檢查的 WSGI 型F . . . 1433
22.2.7 範例 . . . 1434
22.3 urllib --- URL 處理模組 . . . 1435
22.4 urllib.request --- 用來開F URLs 的可擴充函式庫 . . . 1435
22.4.1 Request 物件 . . . 1441
22.4.2 OpenerDirector 物件 . . . 1442
22.4.3 BaseHandler 物件 . . . 1443
22.4.4 HTTPRedirectHandler 物件 . . . 1444
22.4.5 HTTPCookieProcessor 物件 . . . 1445
22.4.6 ProxyHandler 物件 . . . 1445
22.4.7 HTTPPasswordMgr 物件 . . . 1445
22.4.8 HTTPPasswordMgrWithPriorAuth 物件 . . . 1446
22.4.9 AbstractBasicAuthHandler 物件 . . . 1446
22.4.10 HTTPBasicAuthHandler 物件 . . . 1446
22.4.11 ProxyBasicAuthHandler 物件 . . . 1446
22.4.12 AbstractDigestAuthHandler 物件 . . . 1446
22.4.13 HTTPDigestAuthHandler 物件 . . . 1447
22.4.14 ProxyDigestAuthHandler 物件 . . . 1447
22.4.15 HTTPHandler 物件 . . . 1447
22.4.16 HTTPSHandler 物件 . . . 1447
22.4.17 FileHandler 物件 . . . 1447
22.4.18 DataHandler 物件 . . . 1447
22.4.19 FTPHandler 物件 . . . 1447
22.4.20 CacheFTPHandler 物件 . . . 1447
22.4.21 UnknownHandler 物件 . . . 1447
22.4.22 HTTPErrorProcessor 物件 . . . 1448

22.4.23 範例 . . . 1448
22.4.24 遺留介面 . . . 1450
22.4.25 urllib.request 限制 . . . 1451
22.5 urllib.response --- Response classes used by urllib . . . 1452
22.6 urllib.parse --- 將 URL 剖析成元件 . . . 1452
22.6.1 URL Parsing . . . 1453
22.6.2 URL parsing security . . . 1457
22.6.3 Parsing ASCII Encoded Bytes . . . 1458
22.6.4 Structured Parse Results . . . 1458
22.6.5 URL Quoting . . . 1459
22.7 urllib.error --- urllib.request 引發的例外類🄵 . . . 1461
22.8 urllib.robotparser --- robots.txt 的剖析器 . . . 1462
22.9 http --- HTTP 模組 . . . 1463
22.9.1 HTTP 狀態碼 . . . 1463
22.9.2 HTTP 狀態分類 . . . 1465
22.9.3 HTTP 方法 . . . 1466
22.10 http.client --- HTTP 協定用🄵端 . . . 1466
22.10.1 HTTPConnection 物件 . . . 1469
22.10.2 HTTPResponse 物件 . . . 1471
22.10.3 範例 . . . 1472
22.10.4 HTTPMessage 物件 . . . 1473
22.11 ftplib --- FTP 協定用🄵端 . . . 1474
22.11.1 參考 . . . 1474
22.12 poplib --- POP3 協定用🄵端 . . . 1480
22.12.1 POP3 物件 . . . 1481
22.12.2 POP3 範例 . . . 1482
22.13 imaplib --- IMAP4 協定用🄵端 . . . 1483
22.13.1 IMAP4 物件 . . . 1484
22.13.2 IMAP4 範例 . . . 1490
22.14 smtplib --- SMTP 協定用🄵端 . . . 1490
22.14.1 SMTP 物件 . . . 1492
22.14.2 SMTP 範例 . . . 1496
22.15 uuid --- **RFC 9562** 定義的 UUID 物件 . . . 1497
22.15.1 命令列的用法 . . . 1501
22.15.2 範例 . . . 1501
22.15.3 命令列的範例 . . . 1502
22.16 socketserver --- 用於網路伺服器的框架 . . . 1503
22.16.1 Server Creation Notes . . . 1503
22.16.2 Server Objects . . . 1505
22.16.3 Request Handler Objects . . . 1507
22.16.4 範例 . . . 1507
22.17 http.server —HTTP 伺服器 . . . 1511
22.17.1 命令列介面 . . . 1517
22.17.2 安全性注意事項 . . . 1518
22.18 http.cookies --- HTTP 狀態管理 . . . 1518
22.18.1 Cookie 物件 . . . 1519
22.18.2 Morsel 物件 . . . 1520
22.18.3 範例 . . . 1521
22.19 http.cookiejar --- HTTP 用🄵端的 Cookie 處理 . . . 1522
22.19.1 CookieJar 與 FileCookieJar 物件 . . . 1523
22.19.2 FileCookieJar 子類🄵及與網頁🄵覽器的合作 . . . 1525
22.19.3 CookiePolicy 物件 . . . 1526
22.19.4 DefaultCookiePolicy 物件 . . . 1526
22.19.5 Cookie 物件 . . . 1528
22.19.6 範例 . . . 1529
22.20 xmlrpc --- XMLRPC 伺服器與用🄵模組 . . . 1530
22.21 xmlrpc.client --- XML-RPC 用🄵端存取 . . . 1530
22.21.1 ServerProxy 物件 . . . 1532

22.21.2 日期時間物件 . . . 1533
22.21.3 Binary Objects . . . 1534
22.21.4 Fault Objects . . . 1534
22.21.5 ProtocolError 物件 . . . 1535
22.21.6 MultiCall 物件 . . . 1536
22.21.7 便捷的函式 . . . 1536
22.21.8 Example of Client Usage . . . 1537
22.21.9 Example of Client and Server Usage . . . 1537
22.22 xmlrpc.server --- 基本 XML-RPC 伺服器 . . . 1537
22.22.1 SimpleXMLRPCServer 物件 . . . 1538
22.22.2 CGIXMLRPCRequestHandler . . . 1541
22.22.3 Documenting XMLRPC server . . . 1542
22.22.4 DocXMLRPCServer 物件 . . . 1542
22.22.5 DocCGIXMLRPCRequestHandler . . . 1543
22.23 ipaddress --- IPv4/IPv6 操作函式庫 . . . 1543
22.23.1 Convenience factory functions . . . 1543
22.23.2 IP Addresses . . . 1544
22.23.3 IP Network definitions . . . 1549
22.23.4 Interface objects . . . 1554
22.23.5 Other Module Level Functions . . . 1556
22.23.6 Custom Exceptions . . . 1556

23 多媒體服務 1557
23.1 wave --- 讀寫 WAV 檔案 . . . 1557
23.1.1 Wave_read 物件 . . . 1558
23.1.2 Wave_write 物件 . . . 1559
23.2 colorsys --- 🄵色系統間的轉🄵 . . . 1560

24 國際化 1561
24.1 gettext --- 多語言國際化服務 . . . 1561
24.1.1 GNU **gettext** API . . . 1561
24.1.2 Class-based API . . . 1562
24.1.3 Internationalizing your programs and modules . . . 1566
24.1.4 致謝 . . . 1568
24.2 locale --- 國際化服務 . . . 1569
24.2.1 Background, details, hints, tips and caveats . . . 1576
24.2.2 Locale names . . . 1576
24.2.3 For extension writers and programs that embed Python . . . 1577
24.2.4 Access to message catalogs . . . 1577

25 以 Tk 打造圖形使用者介面 (Graphical User Interfaces) 1579
25.1 tkinter --- Tcl/Tk 的 Python 介面 . . . 1579
25.1.1 架構 . . . 1580
25.1.2 Tkinter 模組 . . . 1581
25.1.3 Tkinter 應急指南 . . . 1582
25.1.4 執行緒模型 . . . 1585
25.1.5 實用參考 . . . 1585
25.1.6 檔案處理常式 . . . 1590
25.2 tkinter.colorchooser --- 🄵色選擇對話框 . . . 1591
25.3 tkinter.font --- Tkinter 字型包裝器 . . . 1591
25.4 Tkinter 對話框 . . . 1592
25.4.1 tkinter.simpledialog --- 標准 Tkinter 輸入對話框 . . . 1592
25.4.2 tkinter.filedialog --- File selection dialogs . . . 1593
25.4.3 tkinter.commondialog --- Dialog window templates . . . 1595
25.5 tkinter.messagebox --- Tkinter 訊息提示 . . . 1595
25.6 tkinter.scrolledtext --- 🄵動文字小工具 . . . 1597
25.7 tkinter.dnd --- 拖放支援 . . . 1598
25.8 tkinter.ttk --- Tk 主題化小工具 . . . 1599
25.8.1 使用 Ttk . . . 1599

25.8.2 Ttk Widgets . . . 1599
25.8.3 Widget . . . 1600
25.8.4 Combobox . . . 1602
25.8.5 Spinbox . . . 1603
25.8.6 Notebook . . . 1604
25.8.7 Progressbar . . . 1606
25.8.8 Separator . . . 1606
25.8.9 Sizegrip . . . 1607
25.8.10 Treeview . . . 1607
25.8.11 Ttk Styling . . . 1612
25.9 IDLE --- Python 編輯器與 shell . . . 1617
25.9.1 目F . . . 1617
25.9.2 Editing and Navigation . . . 1621
25.9.3 Startup and Code Execution . . . 1624
25.9.4 Help and Preferences . . . 1627
25.9.5 idlelib --- implementation of IDLE application . . . 1628
25.10 turtle --- 龜圖學 (Turtle graphics) . . . 1628
25.10.1 介紹 . . . 1628
25.10.2 Get started . . . 1628
25.10.3 教學 . . . 1629
25.10.4 How to... . . . 1631
25.10.5 Turtle graphics reference . . . 1632
25.10.6 Methods of RawTurtle/Turtle and corresponding functions . . . 1635
25.10.7 Methods of TurtleScreen/Screen and corresponding functions . . . 1652
25.10.8 Public classes . . . 1659
25.10.9 解釋 . . . 1660
25.10.10Help and configuration . . . 1661
25.10.11turtledemo --- Demo scripts . . . 1663
25.10.12Changes since Python 2.6 . . . 1664
25.10.13Changes since Python 3.0 . . . 1664

26 開發工具 **1667**
26.1 typing --- 支援型F提示 . . . 1667
26.1.1 Python 型F系統的技術規範 . . . 1668
26.1.2 型FF名 . . . 1668
26.1.3 NewType . . . 1669
26.1.4 F釋 callable 物件 . . . 1670
26.1.5 泛型 . . . 1671
26.1.6 F釋元組 (tuple) . . . 1671
26.1.7 類F物件的型F . . . 1672
26.1.8 Annotating generators and coroutines . . . 1673
26.1.9 使用者定義泛型型F . . . 1674
26.1.10 Any 型F . . . 1677
26.1.11 標稱 (nominal) 子型F vs 結構子型F . . . 1678
26.1.12 模組F容 . . . 1678
26.1.13 主要功能的F用時程表 . . . 1720
26.2 pydoc --- 文件F生器與F上幫助系統 . . . 1720
26.3 Python 開發模式 . . . 1721
26.3.1 Python 開發模式的影響 . . . 1721
26.3.2 ResourceWarning 範例 . . . 1722
26.3.3 檔案描述器的錯誤範例 . . . 1723
26.4 doctest --- 測試互動式 Python 範例 . . . 1724
26.4.1 Simple Usage: Checking Examples in Docstrings . . . 1726
26.4.2 Simple Usage: Checking Examples in a Text File . . . 1726
26.4.3 命令列用法 . . . 1727
26.4.4 運作方式 . . . 1727
26.4.5 基礎 API . . . 1734
26.4.6 Unittest API . . . 1736

26.4.7 Advanced API . . . 1738
26.4.8 偵錯 . . . 1743
26.4.9 Soapbox . . . 1746
26.5 unittest --- 單元測試框架 . . . 1747
26.5.1 簡單範例 . . . 1747
26.5.2 命令列介面 . . . 1748
26.5.3 Test Discovery（測試探索） . . . 1750
26.5.4 Organizing test code . . . 1751
26.5.5 Re-using old test code . . . 1752
26.5.6 Skipping tests and expected failures . . . 1753
26.5.7 Distinguishing test iterations using subtests . . . 1754
26.5.8 類🄵與函式 . . . 1755
26.5.9 Class and Module Fixtures . . . 1774
26.5.10 Signal Handling . . . 1776
26.6 unittest.mock —mock 物件函式庫 . . . 1777
26.6.1 快速導引 . . . 1777
26.6.2 Mock 類🄵 . . . 1779
26.6.3 Patchers . . . 1795
26.6.4 MagicMock 以及魔術方法支援 . . . 1804
26.6.5 輔助函式 . . . 1807
26.6.6 side_effect、return_value 和 *wraps* 的優先順序 . . . 1814
26.7 unittest.mock --- 入門指南 . . . 1816
26.7.1 使用 Mock 的方式 . . . 1816
26.7.2 Patch 裝飾器 . . . 1821
26.7.3 更多範例 . . . 1822
26.8 test --- Python 的回歸測試 (regression tests) 套件 . . . 1834
26.8.1 撰寫 test 套件的單元測試 . . . 1834
26.8.2 使用命令列介面執行測試 . . . 1836
26.9 test.support --- Python 測試套件的工具 . . . 1836
26.10 test.support.socket_helper --- 用於 socket 測試的工具 . . . 1846
26.11 test.support.script_helper --- 用於 Python 執行測試的工具 . . . 1846
26.12 test.support.bytecode_helper --- 用於測試位元組碼能正確🄵生的支援工具 . . . 1847
26.13 test.support.threading_helper --- Utilities for threading tests . . . 1848
26.14 test.support.os_helper --- 用於 os 測試的工具 . . . 1849
26.15 test.support.import_helper --- 用於 import 測試的工具 . . . 1851
26.16 test.support.warnings_helper --- 用於 warnings 測試的工具 . . . 1852

27 除錯與效能分析 **1855**
27.1 稽核事件表 . . . 1855
27.2 bdb --- 偵錯器框架 . . . 1859
27.3 faulthandler --- 傾印 Python 回溯 . . . 1864
27.3.1 Dumping the traceback . . . 1865
27.3.2 Dumping the C stack . . . 1865
27.3.3 Fault handler state . . . 1866
27.3.4 Dumping the tracebacks after a timeout . . . 1866
27.3.5 Dumping the traceback on a user signal . . . 1867
27.3.6 Issue with file descriptors . . . 1867
27.3.7 範例 . . . 1867
27.4 pdb --- Python 偵錯器 . . . 1868
27.4.1 偵錯器命令 . . . 1872
27.5 Python 的分析器 . . . 1877
27.5.1 分析器簡介 . . . 1878
27.5.2 Instant User's Manual . . . 1878
27.5.3 profile 與 cProfile 模組參考文件 . . . 1880
27.5.4 Stats 類🄵 . . . 1881
27.5.5 What Is Deterministic Profiling? . . . 1884
27.5.6 限制 . . . 1884
27.5.7 校正 . . . 1885

27.5.8 Using a custom timer . . . 1885
27.6 timeit --- 測量小量程式片段的執行時間 . . . 1886
27.6.1 基礎範例 . . . 1886
27.6.2 Python 介面 . . . 1886
27.6.3 命令列介面 . . . 1888
27.6.4 範例 . . . 1889
27.7 trace --- 追Ⓕ或追查 Python 陳述式執行 . . . 1890
27.7.1 命令列用法 . . . 1891
27.7.2 程式介面 . . . 1892
27.8 tracemalloc --- 追Ⓕ記憶體配置 . . . 1893
27.8.1 範例 . . . 1893
27.8.2 API . . . 1897

28 軟體封裝與發布 1903
28.1 ensurepip --- pip 安裝器的初始建置 (bootstrapping) . . . 1903
28.1.1 命令列介面 . . . 1904
28.1.2 模組 API . . . 1904
28.2 venv --- 建立Ⓕ擬環境 . . . 1905
28.2.1 建立Ⓕ擬環境 . . . 1905
28.2.2 Ⓕ擬環境如何運作 . . . 1907
28.2.3 API . . . 1908
28.2.4 一個擴展 EnvBuilder 的範例 . . . 1910
28.3 zipapp —-- 管理可執行的 Python zip 封存檔案 . . . 1914
28.3.1 基本範例 . . . 1914
28.3.2 命令列介面 . . . 1914
28.3.3 Python API . . . 1915
28.3.4 範例 . . . 1916
28.3.5 指定直譯器 . . . 1916
28.3.6 使用 zipapp 建立獨立應用程式 . . . 1917
28.3.7 Python Zip 應用程式封存檔案格式 . . . 1917

29 Python Runtime 服務 1919
29.1 sys --- 系統特定的參數與函式 . . . 1919
29.2 sys.monitoring --- 執行事件監控 . . . 1946
29.2.1 工具識Ⓔ器 . . . 1947
29.2.2 事件 . . . 1947
29.2.3 開Ⓔ和關閉事件 . . . 1949
29.2.4 ⒻⒻ回呼函式 . . . 1950
29.3 sysconfig --- 提供 Python 設定資訊的存取 . . . 1951
29.3.1 Configuration variables . . . 1951
29.3.2 Installation paths . . . 1952
29.3.3 User scheme . . . 1953
29.3.4 Home scheme . . . 1953
29.3.5 Prefix scheme . . . 1954
29.3.6 安裝路徑函式 . . . 1955
29.3.7 其他函式 . . . 1956
29.3.8 命令列用法 . . . 1957
29.4 builtins --- Ⓕ建物件 . . . 1957
29.5 __main__ --- 頂層程式碼環境 . . . 1958
29.5.1 __name__ == '__main__' . . . 1958
29.5.2 Python 套件中的 __main__.py . . . 1960
29.5.3 import __main__ . . . 1961
29.6 warnings --- 警告控制 . . . 1962
29.6.1 警告類Ⓔ . . . 1963
29.6.2 警告過濾器 . . . 1963
29.6.3 暫時抑制警告 . . . 1965
29.6.4 測試警告 . . . 1966
29.6.5 Ⓕ新版相依套件更新程式碼 . . . 1966

29.6.6 可用的函式 . . . 1966
29.6.7 可用的情境管理器 . . . 1968
29.6.8 情境管理器的🄴行安全性 . . . 1969
29.7 dataclasses --- Data Classes . . . 1969
29.7.1 模組🄴容 . . . 1970
29.7.2 Post-init processing . . . 1976
29.7.3 類🄴變數 . . . 1977
29.7.4 Init-only variables . . . 1977
29.7.5 凍結實例 . . . 1977
29.7.6 繼承 . . . 1977
29.7.7 Re-ordering of keyword-only parameters in __init__() . . . 1978
29.7.8 預設工廠函式 . . . 1978
29.7.9 可變預設值 . . . 1978
29.7.10 Descriptor-typed fields . . . 1979
29.8 contextlib --- with 陳述式工具程式 . . . 1980
29.8.1 Utilities . . . 1980
29.8.2 Examples and Recipes . . . 1989
29.8.3 Single use, reusable and reentrant context managers . . . 1992
29.9 abc --- 抽象基底類🄴 . . . 1994
29.10 atexit --- 退出處理函式 . . . 1999
29.10.1 atexit 範例 . . . 1999
29.11 traceback --- 列印或取得堆🄴回溯 (stack traceback) . . . 2000
29.11.1 Module-Level Functions . . . 2001
29.11.2 TracebackException 物件 . . . 2003
29.11.3 StackSummary 物件 . . . 2005
29.11.4 FrameSummary 物件 . . . 2005
29.11.5 Examples of Using the Module-Level Functions . . . 2006
29.11.6 TracebackException 的使用範例 . . . 2008
29.12 __future__ --- Future 陳述式定義 . . . 2010
29.12.1 模組🄴容 . . . 2010
29.13 gc --- 垃圾回收器介面 (Garbage Collector interface) . . . 2012
29.14 inspect --- 檢視活動物件 . . . 2016
29.14.1 Types and members . . . 2016
29.14.2 取得原始碼 . . . 2021
29.14.3 Introspecting callables with the Signature object . . . 2022
29.14.4 類🄴與函式 . . . 2027
29.14.5 直譯器堆🄴 . . . 2029
29.14.6 Fetching attributes statically . . . 2031
29.14.7 Current State of Generators, Coroutines, and Asynchronous Generators . . . 2032
29.14.8 Code Objects Bit Flags . . . 2033
29.14.9 Buffer flags . . . 2034
29.14.10命令列介面 . . . 2035
29.15 annotationlib --- Functionality for introspecting annotations . . . 2035
29.15.1 Annotation semantics . . . 2036
29.15.2 Classes . . . 2036
29.15.3 Functions . . . 2038
29.15.4 Recipes . . . 2040
29.15.5 Limitations of the STRING format . . . 2041
29.15.6 Limitations of the FORWARDREF format . . . 2042
29.15.7 Security implications of introspecting annotations . . . 2043
29.16 site --- Site-specific configuration hook . . . 2043
29.16.1 sitecustomize . . . 2044
29.16.2 usercustomize . . . 2044
29.16.3 Readline configuration . . . 2045
29.16.4 模組🄴容 . . . 2045
29.16.5 命令列介面 . . . 2046

30 自訂 Python 直譯器 **2047**

30.1 code --- 直譯器基底類🅴 . . . 2047
30.1.1 互動式直譯器物件 . . . 2048
30.1.2 互動式控制台物件 . . . 2048
30.2 codeop --- 編譯 Python 程式碼 . . . 2049

31 引入模組 2051
31.1 zipimport --- 從 Zip 封存檔案匯入模組 . . . 2051
31.1.1 zipimporter 物件 . . . 2052
31.1.2 範例 . . . 2053
31.2 pkgutil --- 套件擴充工具程式 . . . 2053
31.3 modulefinder --- 搜尋🅵本所使用的模組 . . . 2055
31.3.1 ModuleFinder 的用法範例 . . . 2056
31.4 runpy --- 定位🅴執行 Python 模組 . . . 2057
31.5 importlib --- import 的實作 . . . 2059
31.5.1 簡介 . . . 2059
31.5.2 函式 . . . 2060
31.5.3 importlib.abc -- Abstract base classes related to import . . . 2062
31.5.4 importlib.machinery -- Importers and path hooks . . . 2068
31.5.5 importlib.util -- Utility code for importers . . . 2073
31.5.6 範例 . . . 2076
31.6 importlib.resources -- 套件資源的讀取、開🅴與存取 . . . 2079
31.6.1 Functional API . . . 2080
31.7 importlib.resources.abc -- 資源的抽象基底類🅴 . . . 2082
31.8 importlib.metadata -- 存取套件的元資料 . . . 2083
31.8.1 Overview . . . 2084
31.8.2 Functional API . . . 2085
31.8.3 Distributions . . . 2088
31.8.4 Distribution Discovery . . . 2089
31.8.5 Implementing Custom Providers . . . 2089
31.9 sys.path 模組搜尋路徑的初始化 . . . 2091
31.9.1 🅴擬環境 . . . 2092
31.9.2 _pth 檔案 . . . 2092
31.9.3 Embedded Python . . . 2092

32 Python 語言服務 2093
32.1 ast --- 抽象語法樹 (Abstract Syntax Trees) . . . 2093
32.1.1 抽象文法 (Abstract Grammar) . . . 2093
32.1.2 節點 (Node) 類🅵 . . . 2096
32.1.3 ast 輔助程式 . . . 2124
32.1.4 編譯器旗標 . . . 2128
32.1.5 命令列用法 . . . 2129
32.2 symtable --- 存取編譯器的符號表 . . . 2130
32.2.1 🅴生符號表 . . . 2130
32.2.2 檢查符號表 . . . 2130
32.2.3 命令列用法 . . . 2133
32.3 token --- 與 Python 剖析樹一起使用的常數 . . . 2134
32.4 keyword --- 檢驗 Python 關鍵字 . . . 2139
32.5 tokenize --- Tokenizer for Python source . . . 2139
32.5.1 Tokenizing Input . . . 2139
32.5.2 Command-Line Usage . . . 2141
32.5.3 範例 . . . 2141
32.6 tabnanny --- 偵測不良縮排 . . . 2143
32.7 pyclbr --- Python 模組🅴覽器支援 . . . 2144
32.7.1 函式物件 . . . 2144
32.7.2 類🅴物件 . . . 2145
32.8 py_compile —編譯 Python 來源檔案 . . . 2145
32.8.1 命令列介面 . . . 2147
32.9 compileall --- 位元組編譯 Python 函式庫 . . . 2147

32.9.1 Command-line use . . . 2147
32.9.2 Public functions . . . 2149
32.10 dis --- Python bytecode 的反組譯器 . . . 2151
32.10.1 命令列介面 . . . 2152
32.10.2 Bytecode analysis . . . 2152
32.10.3 分析函式 . . . 2153
32.10.4 Python 位元組碼指令 . . . 2156
32.10.5 Opcode collections . . . 2173
32.11 pickletools --- pickle 開發者的工具 . . . 2174
32.11.1 命令列用法 . . . 2174
32.11.2 程式化介面 . . . 2175

33 MS Windows 特有服務 2177
33.1 msvcrt --- MS VC++ runtime 提供的有用例程 . . . 2177
33.1.1 File Operations . . . 2177
33.1.2 Console I/O . . . 2178
33.1.3 Other Functions . . . 2178
33.2 winreg --- Windows 🄵🄵表存取 . . . 2180
33.2.1 函式 . . . 2180
33.2.2 常數 . . . 2185
33.2.3 Registry Handle Objects . . . 2187
33.3 winsound --- Windows 的音效播放介面 . . . 2188

34 Unix 特有服務 2191
34.1 shlex --- 簡單的語法分析 . . . 2191
34.1.1 shlex 物件 . . . 2193
34.1.2 Parsing Rules . . . 2195
34.1.3 Improved Compatibility with Shells . . . 2195
34.2 posix --- 最常見的 POSIX 系統呼叫 . . . 2196
34.2.1 對大檔案 (Large File) 的支援 . . . 2196
34.2.2 值得注意的模組🄵容 . . . 2197
34.3 pwd --- 密碼資料庫 . . . 2197
34.4 grp --- 群組資料庫 . . . 2198
34.5 termios --- POSIX 風格 tty 控制 . . . 2199
34.5.1 範例 . . . 2200
34.6 tty --- 終端機控制函式 . . . 2200
34.7 pty --- 🄵終端工具 . . . 2201
34.7.1 範例 . . . 2202
34.8 fcntl --- fcntl 和 ioctl 系統呼叫 . . . 2203
34.9 resource --- 資源使用資訊 . . . 2206
34.9.1 Resource Limits . . . 2206
34.9.2 Resource Usage . . . 2208
34.10 syslog --- Unix syslog 函式庫例程 . . . 2210
34.10.1 範例 . . . 2212

35 模組命令列介面 2213

36 已被取代的模組 2215
36.1 getopt --- 用於命令列選項的 C 風格剖析器 . . . 2215

37 已移除的模組 2219
37.1 aifc --- 讀寫 AIFF 與 AIFC 檔案 . . . 2219
37.2 asynchat --- 非同步 socket 指令/回應處理函式 . . . 2219
37.3 asyncore --- 非同步 socket 處理函式 . . . 2219
37.4 audioop --- 操作原始聲音檔案 . . . 2220
37.5 cgi --- 通用閘道器介面支援 . . . 2220
37.6 cgitb --- CGI 🄵本的回溯管理器 (traceback manager) . . . 2220
37.7 chunk --- 讀取 IFF 分塊資料 . . . 2220
37.8 crypt --- 用於檢查 Unix 密碼的函式 . . . 2220

37.9 distutils --- 建置與安裝 Python 模組 . . . 2220
37.10 imghdr --- 判定圖片種類 . . . 2221
37.11 imp --- 存取引入系統🄴層 . . . 2221
37.12 mailcap --- Mailcap 檔案處理 . . . 2221
37.13 msilib --- 讀寫 Microsoft Installer 檔案 . . . 2221
37.14 nis --- Sun NIS (Yellow Pages) 介面 . . . 2221
37.15 nntplib --- NNTP 協定用🄴端 . . . 2221
37.16 ossaudiodev --- 對 OSS 相容聲音裝置的存取 . . . 2222
37.17 pipes --- shell pipelines 介面 . . . 2222
37.18 smtpd --- SMTP 伺服器 . . . 2222
37.19 sndhdr --- 判定聲音檔案的種類 . . . 2222
37.20 spwd --- shadow 密碼資料庫 . . . 2222
37.21 sunau --- 讀寫 Sun AU 檔案 . . . 2222
37.22 telnetlib --- Telnet 用🄴端 . . . 2223
37.23 uu --- uuencode 檔案的編碼與解碼 . . . 2223
37.24 xdrlib --- XDR 資料的編碼與解碼 . . . 2223

38 安全性注意事項 **2225**

A 術語表 **2227**

B 關於這份🄴明文件 **2245**
B.1 Python 文件的貢獻者們 . . . 2245

C 沿革與授權 **2247**
C.1 軟體沿革 . . . 2247
C.2 關於存取或以其他方式使用 Python 的合約條款 . . . 2248
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 2248
C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 . . . 2249
C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 . . . 2249
C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 . . . 2250
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . 2251
C.3 被收🄴軟體的授權與致謝 . . . 2251
C.3.1 Mersenne Twister . . . 2251
C.3.2 Sockets . . . 2252
C.3.3 非同步 socket 服務 . . . 2252
C.3.4 Cookie 管理 . . . 2253
C.3.5 執行追🄴 . . . 2253
C.3.6 UUencode 與 UUdecode 函式 . . . 2254
C.3.7 XML 遠端程序呼叫 . . . 2254
C.3.8 test_epoll . . . 2255
C.3.9 Select kqueue . . . 2255
C.3.10 SipHash24 . . . 2256
C.3.11 strtod 與 dtoa . . . 2256
C.3.12 OpenSSL . . . 2256
C.3.13 expat . . . 2259
C.3.14 libffi . . . 2260
C.3.15 zlib . . . 2260
C.3.16 cfuhash . . . 2261
C.3.17 libmpdec . . . 2261
C.3.18 W3C C14N 測試套件 . . . 2262
C.3.19 mimalloc . . . 2263
C.3.20 asyncio . . . 2263
C.3.21 Global Unbounded Sequences (GUS) . . . 2263
C.3.22 Zstandard bindings . . . 2264

D 版權宣告 **2265**

Bibliography **2267**

Python 模組索引 2269

索引 2273

reference-index 🄵明 Python 這門語言確切的文法及語意，而這份函式庫參考手🄵則是🄵明隨著 Python 一起發🄵的標准函式庫，除此之外，其🄵容也包含一些時常出現在 Python 發🄵版本中的非必要套件。

Python 的標准函式庫是非常龐大的，其提供了如下所述極多且涵蓋用途極廣的許多模組。包含一些用 C 語言撰寫，可以操作像是檔案讀寫等系統相關功能的🄵建模組，當然也有用 Python 撰寫，🄵使用標准解法解🄵許多常見問題的模組。其中有些模組則是特🄵針對 Python 的可🄵性去設計的，🄵此特地將一些平台特殊相依性的功能抽象化成可跨平台的 API。

Python 的 Windows 安裝檔基本上包含整個標准函式庫，且通常也包含許多附加的組件；而在類 Unix 作業系統方面，Python 通常是以一系列的套件被安裝，因此對於某些或全部的可選組件，可能都必須使用該作業系統提供的套件管理工具來安裝。

在標准函式庫之外，還有成千上萬且不斷增加的組件（從個🄵的程式、模組、套件到完整的應用程式開發框架），可以從 Python 套件索引 (Python Package Index) 中取得。

CHAPTER 1

簡介

「Python 函式庫」包含了許多不同的部分。

函式庫中包括被視🄵程式語言「核心」部分的資料型態，像是數字 (number) 或是串列 (list)。對於這些型🄵，Python 核心對這些字面值 (literal) 的形式做定義，🄵對它們的語意制定了一些限制，但在此同時🄵不把文字對應的語意完全定義。（另一方面，Python 在語法面上有確實的定義，例如拼字或是運算元次序）

Python 函式庫也囊括了🄵建函式與例外處理——這些物件都可以不用透過 `import` 陳述式來引入 Python 程式中就能使用。函式庫中有部份是被 Python 核心所定義的，但在這🄵僅解釋最核心的語意部分。

整個函式庫中包含了許多模組，有許多方法可以從函式庫中取用這些模組。有些模組是以 C 語言撰寫🄵建置於 Python 編譯器之中，其他的是由 Python 撰寫🄵以源碼的方式 (source form) 引入。有些模組提供的功能是專屬於 Python 的，像是把 stack trace 印出來；有些則是針對特定作業系統，去試著存取特定硬體；還有些提供對特定應用的功能與操作介面，像是 World Wide Web。模組的使用情🄵會因機器與 Python 的版本而不同，部分模組是開放所有版本以及 Port 的 Python 來使用的，但有些會因系統支援或需求在某些版本或系統下無法使用，甚至有些僅限在特定的設定環境下才能使用。

這個手🄵會「深入淺出」地介紹 Python 函式庫。它會先介紹一些🄵建函式、資料型態、和一些例外處理，再來一章章的主題式介紹相關模組。

這代表如果你從這個手🄵的最開始讀起，🄵在感到無聊時跳到下一個章節，你仍然可以得到一個對 Python 函式庫所支援的模組與其合理應用的概觀。當然，你🄵有必要像是在讀一本小🄵一樣讀這本手🄵——你可以快速🄵覽目🄵（在手🄵的最前頭）、或是你可以利用最後面的索引來查詢特定的函式或模組。最後，如果你享受🄵讀一些隨機的主題，你可以選擇一個隨機的數字🄵開始🄵讀（見 *random* 模組）。不管你想要以什🄵順序來🄵讀這個手🄵，🄵建函式會是一個很好的入門，因🄵手🄵中其他章節都預設你已經對這個章節有一定的熟悉程度。

讓我們開始吧！

1.1 可用性之標🄵

- 如果出現「適用：Unix」標🄵，則代表該函式普遍存在於 Unix 系統中，但這🄵不保證其存在於某特定作業系統。
- 如果🄵有分🄵標🄵的話，有標明「適用：Unix」標🄵的所有函式也都於 macOS、iOS 和 Android 上支援，因其建於 Unix 核心之上。
- If an availability note contains both a minimum Kernel version and a minimum libc version, then both conditions must hold. For example a feature with note *Availability: Linux >= 3.17 with glibc >= 2.27* requires both Linux 3.17 or newer and glibc 2.27 or newer.

1.1.1 WebAssembly 平台

The WebAssembly platforms `wasm32-emscripten` (Emscripten) and `wasm32-wasi` (WASI) provide a subset of POSIX APIs. WebAssembly runtimes and browsers are sandboxed and have limited access to the host and external resources. Any Python standard library module that uses processes, threading, networking, signals, or other forms of inter-process communication (IPC), is either not available or may not work as on other Unix-like systems. File I/O, file system, and Unix permission-related functions are restricted, too. Emscripten does not permit blocking I/O. Other blocking operations like *`sleep()`* block the browser event loop.

The properties and behavior of Python on WebAssembly platforms depend on the Emscripten-SDK or WASI-SDK version, WASM runtimes (browser, NodeJS, wasmtime), and Python build time flags. WebAssembly, Emscripten, and WASI are evolving standards; some features like networking may be supported in the future.

For Python in the browser, users should consider Pyodide or PyScript. PyScript is built on top of Pyodide, which itself is built on top of CPython and Emscripten. Pyodide provides access to browsers' JavaScript and DOM APIs as well as limited networking capabilities with JavaScript's `XMLHttpRequest` and `Fetch` APIs.

- Process-related APIs are not available or always fail with an error. That includes APIs that spawn new processes (*`fork()`*, *`execve()`*), wait for processes (*`waitpid()`*), send signals (*`kill()`*), or otherwise interact with processes. The *`subprocess`* is importable but does not work.
- The *`socket`* module is available, but is limited and behaves differently from other platforms. On Emscripten, sockets are always non-blocking and require additional JavaScript code and helpers on the server to proxy TCP through WebSockets; see Emscripten Networking for more information. WASI snapshot preview 1 only permits sockets from an existing file descriptor.
- Some functions are stubs that either don't do anything and always return hardcoded values.
- Functions related to file descriptors, file permissions, file ownership, and links are limited and don't support some operations. For example, WASI does not permit symlinks with absolute file names.

1.1.2 行動平台

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

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

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

CHAPTER 2

🄵建函式

Python 直譯器有🄵建多個可隨時使用的函式和型🄵。以下按照英文字母排序列出。

🄵建函式

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

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

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

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

E
enumerate()
eval()
exec()

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

G
getattr()
globals()

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

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

L
len()
list()
locals()

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

N
next()

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

P
pow()
print()
property()

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

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

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*number*, */ (Positional-only parameter separator (PEP 570))*)

回傳一個數的🄵對值，引數可以是整數、浮點數或有實現 `__abs__()` 的物件。如果引數是一個🄵數，回傳它的純量（大小）。

aiter(*async_iterable*, */*)

回傳非同步🄵代器 做🄵非同步可🄵代物件。相當於呼叫 x.__aiter__()。

注意：與*iter()* 不同，*aiter()* 🄵有兩個引數的變體。

在 3.10 版被加入.

all(*iterable*, */*)

如果 *iterable* 的所有元素皆🄵真（或 iterable 🄵空）則回傳 `True`。等價於：

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

awaitable anext(*async_iterator*, */*)

awaitable anext(*async_iterator*, *default*, /)

當進入 await 時，從給定的*asynchronous iterator* 中回傳下一個項目（item），🄵代完畢則回傳 *default* 。

這是🄵建函式*next()* 的非同步版本，其行🄵類似於：

呼叫 *async_iterator* 的 `__anext__()` 方法，回傳*awaitable*。等待返回🄵代器的下一個值。如果指定 *default*，當🄵代器結束時會返回該值，否則會引發*StopAsyncIteration* 。

在 3.10 版被加入.

any(*iterable*, /)

如果 *iterable* 的任一元素🄵真，回傳 `True`。如果 iterable 是空的，則回傳 `False`。等價於：

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

ascii(*object*, /)

就像函式*repr()*，回傳一個表示物件的字串，但是*repr()* 回傳的字串中非 ASCII 編碼的字元會被跳🄵 (escape)，像是 `\x`、`\u` 和 `\U`。這個函式生成的字串和 Python 2 的*repr()* 回傳的結果相似。

bin(*integer*, /)

將一個整數轉變🄵一個前綴🄵"0b" 的二進位制字串。結果是一個有效的 Python 運算式。如果 *integer* 不是 Python 的*int* 物件，那它需要定義 `__index__()` method 回傳一個整數。舉例來🄵：

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

如果不一定需要"0b" 前綴，還可以使用如下的方法。

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

可參考*format()* 取得更多資訊。

class bool(*object=False*, /)

回傳一個布林值，即 `True` 或者 `False`。引數會使用標準的*真值測試程序*來轉🄵。如果引數🄵假或者被省略，則回傳 `False`；其他情🄵回傳 `True`。*bool* class（類🄵）是*int* 的 subclass（子類🄵）（參見*數值型🄵 --- int、float、complex*），其他 class 不能繼承自它。它只有 `False` 和 `True` 兩個實例（參見*Boolean 型🄵 - bool*）。

在 3.7 版的變更: 現在🄵僅限位置參數。

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

這個函式將呼叫*sys.breakpointhook()* 函式，🄵將 `args` 和 `kws` 傳遞給它。這將有效地讓你在特定的呼叫點進入除錯器。預設情🄵下，`sys.breakpointhook()` 呼叫*pdb.set_trace()* 不須帶任何引數。這樣的設計是🄵了方便使用者，讓他們不需要額外地導入*pdb* 模組或輸入太多程式就可以進入除錯器。然而，可以將*sys.breakpointhook()* 設置🄵其他函式，🄵且*breakpoint()* 將自動呼叫該函式，讓你進入所選擇的除錯器。如果無法存取*sys.breakpointhook()* 這個函式，則此函式將引發*RuntimeError*。

預設情🄵下，*breakpoint()* 的行🄵可以透過 `PYTHONBREAKPOINT` 環境變數來更改。有關使用詳情，請參考*sys.breakpointhook()*。

請注意，如果*sys.breakpointhook()* 被替🄵了，則無法保證此功能。

引發一個附帶引數 `breakpointhook` 的*稽核事件* `builtins.breakpoint`。

在 3.7 版被加入.

class bytearray(*source=b''*)

class bytearray(*source*, *encoding*, *errors='strict'*)

回傳一個新的 bytes 陣列。*bytearray* class 是一個可變的整數序列，包含範圍🄵 0 <= x < 256 的整數。它有可變序列大部分常見的 method（如在可變序列型🄵 中所述），同時也有*bytes* 型🄵大部分的 method，參見*Bytes 和 Bytearray 的操作*。

選擇性參數 *source* 可以被用來以不同的方式初始化陣列：

- 如果是一個 *string*，你必須提供 *encoding* 參數（以及選擇性地提供 *errors* ）；*bytearray()* 會使用*str.encode()* method 來將 string 轉變成 bytes。
- 如果是一個 *integer*，陣列則會有該數值的長度，🄵以 null bytes 來當作初始值。
- 如果是一個符合 buffer 介面的物件，該物件的唯讀 buffer 會被用來初始化 bytes 陣列。
- 如果是一個 *iterable*，它的元素必須是範圍🄵 `0 <= x < 256` 的整數，🄵且會被用作陣列的初始值。

如果🄵有引數，則建立長度🄵 0 的陣列。

可參考*Binary Sequence Types --- bytes, bytearray, memoryview* 和*Bytearray 物件*。

class bytes(*source=b''*)

class bytes(*source*, *encoding*, *errors='strict'*)

回傳一個新的"bytes" 物件，會是一個元素是範圍🄵 `0 <= x < 256` 整數的不可變序列。*bytes* 是*bytearray* 的不可變版本—它的同樣具備不改變物件的 method，也有相同的索引和切片操作。

因此，建構函式的引數和*bytearray()* 相同。

Bytes 物件還可以用文字建立，參見 strings。

可參考*Binary Sequence Types --- bytes, bytearray, memoryview*、*Bytes Objects* 和*Bytes 和 Bytearray 的操作*。

callable(*object*, */*)

如果引數 *object* 是可呼叫的，回傳*True*，否則回傳*False*。如果回傳 `True`，呼叫仍可能會失敗；但如果回傳 `False`，則呼叫 *object* 肯定會失敗。注意 class 是可呼叫的（呼叫 class 會回傳一個新的實例）；如果實例的 class 有定義 `__call__()` method，則它是可呼叫的。

在 3.2 版被加入: 這個函式一開始在 Python 3.0 被移除，但在 Python 3.2 又被重新加入。

chr(*codepoint*, */*)

回傳代表有特定 Unicode 編碼位置字元的字串。例如，`chr(97)` 回傳字串 `'a'`，而 `chr(8364)` 回傳字串 `'€'`。這是*ord()* 的逆函式。

引數的有效範圍是 0 到 1,114,111（16 進制表示🄵 0x10FFFF）。如果它超過這個範圍，會引發*ValueError*。

@classmethod

把一個 method 封裝成 class method（類🄵方法）。

一個 class method 把自己的 class 作🄵第一個引數，就像一個實例 method 把實例自己作🄵第一個引數。請用以下慣例來宣告 class method：

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

`@classmethod` 語法是一個函式*decorator* —參見 function 中關於函式定義的詳細介紹。

一個 class method 可以在 class（如 `C.f()`）或實例（如 `C().f()`）上呼叫。實例除了它的 class 資訊，其他都會被忽略。如果一個 class method 在 subclass 上呼叫，subclass 會作🄵第一個引數傳入。

Class method 和 C++ 與 Java 的 static method 是有區🄵的。如果你想🄵解 static method，請看本節的*staticmethod()*。關於 class method 的更多資訊，請參考 types。

在 3.9 版的變更: Class methods 現在可以包裝其他描述器，例如*property()*

在 3.10 版的變更: Class method 現在繼承了 method 屬性（`__module__`、`__name__`、`__qualname__`、`__doc__` 和 `__annotations__`），🄵擁有一個新的 `__wrapped__` 屬性。

Deprecated since version 3.11, removed in version 3.13: Class methods 不能再包裝其他的描述器，例如*property()*。

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

將 *source* 編譯成程式碼或 AST 物件。程式碼物件可以被*exec()* 或*eval()* 執行。*source* 可以是一般的字串、bytes 字串、或者 AST 物件。參見*ast* module（模組）的🄵明文件🄵解如何使用 AST 物件。

filename 引數必須是程式碼的檔名；如果程式碼不是從檔案中讀取，可以傳入一些可辨識的值（經常會使用 `'<string>'` 來替代）。

mode 引數指定了編譯程式碼時必須用的模式。如果 *source* 是一系列的陳述式，可以是 `'exec'`；如果是單一運算式，可以是 `'eval'`；如果是單個互動式陳述式，可以是 `'single'`（在最後一種情🄵下，如果運算式執行結果不是 `None` 則會被印出來）。

可選引數 *flags* 和 *dont_inherit* 控制🄵用哪個編譯器選項以及允許哪個未來功能。如果兩者都不存在（或兩者都🄵零），則會呼叫與*compile()* 相同旗標的程式碼來編譯。如果給定 *flags* 引數而未給定 *dont_inherit** （或🄵零） 則無論如何都會使用由 **flags* 引數所指定的編譯器選項和未來陳述式。如果 *dont_inherit* 是一個非零整數，則使用 *flags* 引數 -- 周圍程式碼中的旗標（未來功能和編譯器選項）將被忽略。

編譯器選項和 future 陳述式使用 bits 來表示，可以一起被位元操作 OR 來表示🄵數個選項。需要被具體定義特徵的位元域可以透過*__future__* module 中*_Feature* 實例中的*compiler_flag* 屬性來獲得。編譯器旗標可以在*ast* module 中搜尋有 `PyCF_` 前綴的名稱。

引數 *optimize* 用來指定編譯器的最佳化級🄵；預設值 `-1` 選擇與直譯器的 `-O` 選項相同的最佳化級🄵。其他級🄵🄵 `0`（🄵有最佳化；`__debug__` 🄵真值）、`1`（assert 被🄵除，`__debug__` 🄵假值）或 `2`（🄵明字串 (docstring) 也被🄵除）。

如果編譯的原始碼無效，此函式會引發*SyntaxError*，如果原始碼包含 null bytes，則會引發*ValueError*。

如果你想解析 Python 程式碼🄵 AST 運算式，請參🄵*ast.parse()*。

引發一個附帶引數 `source`、`filename` 的稽核事件 `compile`。此事件也可能由隱式編譯 (implicit compilation) 所引發。

> **備🄵**
>
> 在 `'single'` 或 `'eval'` 模式編譯多行程式碼時，輸入必須以至少一個🄵行符結尾。這使*code* module 更容易檢測陳述式的完整性。

> **警告**
>
> 如果編譯足🄵大或者足🄵🄵雜的字串成 AST 物件時，Python 直譯器會因🄵 Python AST 編譯器的 stack 深度限制而崩潰。

在 3.2 版的變更: 允許使用 Windows 和 Mac 的🄵行符號。此外，在 `'exec'` 模式不需要以🄵行符號結尾。增加了 *optimize* 參數。

在 3.5 版的變更: 在之前的版本，*source* 中包含 null bytes 會引發*TypeError*。

在 3.8 版被加入: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` 現在可以傳遞旗標以🄵用對頂層 `await`、`async for` 和 `async with` 的支援。

class complex(*number=0*, */*)

class complex(*string*, */*)

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

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

例如：

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

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

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

如果引數是一個數字，則建構函式會像*int* 和*float* 一樣進行數值轉🄵。對於一個普通的 Python 物件 `x`，`complex(x)` 會委派給 `x.__complex__()`。如果 `__complex__()` 未定義，則會回退 (fall back) 到 `__float__()`。如果 `__float__()` 未定義，則會再回退到 `__index__()`。

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

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

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

🄵數型🄵在*數值型🄵 --- int*、*float*、*complex* 中有相關描述。

在 3.6 版的變更: 可以使用底🄵將程式碼文字中的數字進行分組。

在 3.8 版的變更: 如果 `__complex__()` 和 `__float__()` 未定義，則會回退到 `__index__()`。

在 3.14 版之後被🄵用: Passing a complex number as the *real* or *imag* argument is now deprecated; it should only be passed as a single positional argument.

delattr(*object*, *name*, */*)

這是*setattr()* 相關的函式。引數是一個物件和一個字串，該字串必須是物件中某個屬性名稱。如果物件允許，該函式將🄵除指定的屬性。例如 `delattr(x, 'foobar')` 等價於 `del x.foobar`。*name* 不必是個 Python 識🄵符 (identifier)（請見*setattr()*）。

class dict(***kwargs*)

class dict(*mapping*, */*, ***kwargs*)

class dict(*iterable*, */*, ***kwargs*)

建立一個新的 dictionary（字典）。*dict* 物件是一個 dictionary class。參見*dict* 和*Mapping Types --- dict* 來F解這個 class。

其他容器型F，請參見F建的*list*、*set* 和*tuple* class，以及*collections* module。

dir()

dir(*object*, */*)

如果F有引數，則回傳目前區域作用域 (local scope) 中的名稱列表。如果有引數，它會嘗試回傳該物件的有效屬性列表。

如果物件有一個名F `__dir__()` 的 method，那F該 method 將被呼叫，F且必須回傳一個屬性列表。這允許實現自定義 `__getattr__()` 或 `__getattribute__()` 函式的物件能F自定義*dir()* 來報告它們的屬性。

如果物件不提供 `__dir__()`，這個函式會嘗試從物件已定義的 `__dict__` 屬性和型F物件收集資訊。結果列表F不總是完整的，如果物件有自定義 `__getattr__()`，那結果可能不准確。

預設的*dir()* 機制對不同型F的物件有不同行F，它會試圖回傳最相關而非最完整的資訊：

- 如果物件是 module 物件，則列表包含 module 的屬性名稱。
- 如果物件是型F或 class 物件，則列表包含它們的屬性名稱，F且遞F查詢其基礎的所有屬性。
- 否則，包含物件的屬性名稱列表、它的 class 屬性名稱，F且遞F查詢它的 class 的所有基礎 class 的屬性。

回傳的列表按字母表排序，例如：

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

> **備F**
>
> 因F*dir()* 主要是F了便於在互動式提示字元時使用，所以它會試圖回傳人們感興趣的名稱集合，而不是試圖保證結果的嚴格性或一致性，它具體的行F也可能在不同版本之間改變。例如，當引數是一個 class 時，metaclass 的屬性不包含在結果列表中。

divmod(*a*, *b*, */*)

它將兩個（非F數）數字作F引數，F在執行整數除法時回傳一對商和余數。對於混合運算元型F，適用二進位算術運算子的規則。對於整數，運算結果和 `(a // b, a % b)` 一致。對於浮點數，運算結果是 `(q, a % b)`，*q* 通常是 `math.floor(a / b)` 但可能會比 1 小。在任何情F下，`q * b + a % b` 和 *a* 基本相等，如果 `a % b` 非零，則它的符號和 *b* 一樣，且 `0 <= abs(a % b) < abs(b)`。

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

回傳一個列舉 (enumerate) 物件。*iterable* 必須是一個序列、*iterator* 或其他支援F代的物件。*enumerate()* 回傳之 iterator 的*__next__()* method 回傳一個 tuple（元組），F面包含一個計數值（從 *start* 開始，預設F 0）和透過F代 *iterable* 獲得的值。

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

等價於：

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

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

參數

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

回傳

The result of the evaluated expression.

引發

Syntax errors are reported as exceptions.

警告

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

expression 引數會被視🄵一條 Python 運算式（技術上而言，是條件列表）來剖析及求值，而 *globals* 和 *locals* 對映分🄵用作全域和區域命名空間。如果 *globals* dictionary 存在但缺少 __builtins__ 的鍵值，那 *expression* 被剖析之前，將🄵該鍵插入對🄵建*builtins* module dictionary 的引用。這🄵一來，在將 __builtins__ dictionary 傳入*eval()* 之前，你可以透過將它插入 *globals* 來控制你需要哪些🄵建函式來執行程式碼。如果 *locals* 對映被省略，那它的預設值是 *globals* dictionary。如果兩個對映都被省略，則以在*eval()* 被呼叫的環境中的 *globals* 和 *locals* 執行運算式。請注意，*eval()* 在封閉 (enclosing) 環境中無法存取巢狀作用域 (non-locals)，除非呼叫*eval()* 的作用域已經有參照它們（例如透過 nonlocal 陳述式）。

範例：

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

這個函式也可以用來執行任意程式碼物件（如被*compile()* 建立的那些）。這種情🄵下，傳入的引數是程式碼物件而不是字串。如果編譯該物件時的 *mode* 引數是 'exec'，那🄵*eval()* 回傳值🄵 None。

提示：*exec()* 函式支援動態執行陳述式。*globals()* 和*locals()* 函式分🄵回傳目前的全域性和局部性 dictionary，它們對於將引數傳遞給*eval()* 或*exec()* 可能會方便許多。

如果給定來源是一個字串，那🄵其前後的空格和定位字元會被移除。

另外可以參🄵*ast.literal_eval()*，該函式可以安全執行僅包含文字的運算式字串。

引發一個附帶程式碼物件🄵引數的稽核事件 exec。也可能會引發程式碼編譯事件。

在 3.13 版的變更: The *globals* and *locals* arguments can now be passed as keywords.

在 3.13 版的變更: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

exec(*source*, */*, *globals=None*, *locals=None*, ** (Keyword-only parameters separator (PEP 3102))*, *closure=None*)

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

這個函式支援動態執行 Python 程式碼。*source* 必須是字串或者程式碼物件。如果是字串，那🄵該字串將被剖析🄵一系列 Python 陳述式🄵執行（除非發生語法錯誤）。[1] 如果是程式碼物件，它將被直接執行。無論哪種情🄵，被執行的程式碼都需要和檔案輸入一樣是有效的（可參🄵語言參考手🄵中關於 file-input 的章節）。請注意，即使在傳遞給`exec()` 函式的程式碼的上下文中，`nonlocal`、`yield` 和 `return` 陳述式也不能在函式之外使用。該函式回傳值是 `None`。

無論哪種情🄵，如果省略了選擇性的部分，程式碼將在目前作用域🄴執行。如果只提供了 *globals* 引數，就必須是 dictionary 型🄵（且不能是 dictionary 的子類🄵），而且會被用作全域和區域變數。如果同時提供了 *globals* 和 *locals*，它們分🄵被用作全域和區域變數。如果提供了 *locals*，則它可以是任何對映物件。請記住在 module 層級中全域和區域變數是相同的 dictionary。

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

如果 *globals* dictionary 不包含 `__builtins__` 鍵值，則將🄵該鍵插入對🄵建`builtins` module dictionary 的引用。這🄵一來，在將 `__builtins__` dictionary 傳入`exec()` 之前，你可以透過將它插入 *globals* 來控制你需要哪些🄵建函式來執行程式碼。

closure 引數會指定一個閉包 (closure)—它是一個 cellvar（格變數）的 tuple。只有在 *object* 是一個含有*自由（閉包）變數 (free (closure) variables)* 的程式碼物件時，它才有效。Tuple 的長度必須與程式碼物件的 `co_freevars` 屬性完全匹配。

引發一個附帶程式碼物件🄵引數的*稽核事件* `exec`。也可能會引發程式碼編譯事件。

> **備🄵**
>
> 🄵建`globals()` 和`locals()` 函式各自回傳目前的全域和區域命名空間，因此可以將它們傳遞給`exec()` 的第二個和第三個引數以供後續使用。

> **備🄵**
>
> 預設情🄵下，*locals* 的行🄵如下面`locals()` 函式描述的一樣。如果你想在`exec()` 函式回傳時知道程式碼對 *locals* 的變動，請明確地傳遞 *locals* dictionary 。

在 3.11 版的變更: 增加了 *closure* 參數。

在 3.13 版的變更: The *globals* and *locals* arguments can now be passed as keywords.

在 3.13 版的變更: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

[1] 剖析器只接受 Unix 風格的行結束符。如果你從檔案中讀取程式碼，請確保用🄵行符號轉🄵模式轉🄵 Windows 或 Mac 風格的🄵行符號。

filter(*function*, *iterable*, /)

用 *iterable* 中函式 *function* 為 True 的那些元素，構建一個新的 iterator。*iterable* 可以是一個序列、一個支援疊代的容器、或一個 iterator。如果 *function* 是 `None`，則會假設它是一個識別性函式，即 *iterable* 中所有假值元素會被移除。

請注意，`filter(function, iterable)` 相當於一個生成器運算式，當 function 不是 `None` 的時候為 `(item for item in iterable if function(item))`；function 是 `None` 的時候為 `(item for item in iterable if item)`。

請參見 *itertools.filterfalse()*，只有 *function* 為 false 時才選取 *iterable* 中元素的互補函式。

class float(*number=0.0*, /)

class float(*string*, /)

回傳從數字或字串生成的浮點數。

例如：

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

如果引數是字串，則它必須是包含十進位制數字的字串，字串前面可以有符號，之前也可以有空格。選擇性的符號有 `'+'` 和 `'-'`；`'+'` 對建立的值有影響。引數也可以是 NaN（非數字）或正負無窮大的字串。確切地說，除去首尾的空格後，輸入必須遵循以下語法中 *floatvalue* 的生成規則：

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

字母大小寫不影響，例如，”inf”、”Inf”、”INFINITY”、”iNfINity” 都可以表示正無窮大。

否則，如果引數是整數或浮點數，則回傳具有相同值（在 Python 浮點精度範圍內）的浮點數。如果引數在 Python 浮點精度範圍外，則會引發 *OverflowError*。

對於一般的 Python 物件 `x`，`float(x)` 會委派給 `x.__float__()`。如果未定義 `__float__()` 則會回退到 `__index__()`。

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

如果有引數，則回傳 `0.0`。

數值型 --- *int*、*float*、*complex* 描述了浮點數型。

在 3.6 版的變更: 可以使用底將程式碼文字中的數字進行分組。

在 3.7 版的變更: 現在僅限位置參數。

在 3.8 版的變更: 如果 `__float__()` 未定義，則會回退到 `__index__()`。

format(*value*, *format_spec*='', /)

將 *value* 轉🄵🄵 *format_spec* 控制的" 格式化" 表示。*format_spec* 的解釋取🄵於 *value* 引數的型🄵，但是大多數🄵建型🄵使用標准格式化語法：格式規格 (Format Specification) 迷你語言。

預設的 *format_spec* 是一個空字串，它通常和呼叫`str(value)` 的效果相同。

呼叫 `format(value, format_spec)` 會轉🄵成 `type(value).__format__(value, format_spec)`，當搜尋 value 的 `__format__()` method 時，會忽略實例中的字典。如果搜尋到`object` 這個 method 但 *format_spec* 不🄵空，或是 *format_spec* 或回傳值不是字串，則會引發`TypeError`。

在 3.4 版的變更：當 *format_spec* 不是空字串時，`object().__format__(format_spec)` 會引發`TypeError`。

class frozenset(*iterable=()*, /)

回傳一個新的`frozenset` 物件，它包含選擇性引數 *iterable* 中的元素。`frozenset` 是一個🄵建的 class。有關此 class 的文件，請參🄵`frozenset` 和*Set Types --- set, frozenset*。

請參🄵🄵建的`set`、`list`、`tuple` 和`dict` class，以及`collections` module 來了解其它的容器。

getattr(*object*, *name*, /)

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

回傳 *object* 之具名屬性的值。*name* 必須是字串。如果該字串是物件屬性之一的名稱，則回傳該屬性的值。例如，`getattr(x, 'foobar')` 等同於 `x.foobar`。如果指定的屬性不存在，且提供了 *default* 值，則回傳其值，否則引發`AttributeError`。*name* 不必是個 Python 識🄵符 (identifier)（請見`setattr()`）。

> **備🄵**
>
> 由於私有名稱改編 (private name mangling) 是發生在編譯期，因此你必須手動改編私有屬性（有兩個前導底🄵的屬性）的名稱，才能使用`getattr()` 來取得它。

globals()

回傳代表目前 module 命名空間的 dictionary。對於在函式中的程式碼來🄵，這在定義函式時設定且不論該函式是在何處呼叫都會保持相同。

hasattr(*object*, *name*, /)

該引數是一個物件和一個字串。如果字串是物件屬性之一的名稱，則回傳 `True`，否則回傳 `False`。(此功能是透過呼叫 `getattr(object, name)` 🄵檢查是否引發`AttributeError` 來實作的。)

hash(*object*, /)

回傳該物件的雜🄵值（如果它有的話）。雜🄵值是整數。它們在 dictionary 查詢元素時用來快速比較 dictionary 的鍵。相同大小的數字數值有相同的雜🄵值（即使它們型🄵不同，如 1 和 1.0）。

> **備🄵**
>
> 請注意，如果物件帶有自訂的 `__hash__()` 方法，`hash()` 將根據運行機器的位元長度來截斷回傳值。

help()

help(*request*)

🄵動🄵建的幫助系統（此函式主要以互動式使用）。如果🄵有引數，直譯器控制台🄵會🄵動互動式幫助系統。如果引數是一個字串，則會在 module、函式、class、method、關鍵字或🄵明文件主題中搜尋該字串，🄵在控制台上列印幫助資訊。如果引數是其他任意物件，則會生成該物件的幫助頁。

請注意，呼叫`help()` 時，如果斜🄵 (/) 出現在函式的參數列表中，這表示斜🄵前面的參數是僅限位置 (positional-only) 參數。有關更多資訊，請參🄵常見問答集中的僅限位置參數條目。

此函式會被`site` module 加入到🄵建命名空間。

在 3.4 版的變更: 對於*pydoc* 和*inspect* 的變更，使得可呼叫物件回報的的簽名 (signature) 更加全面和一致。

hex(*integer*, /)

將整數轉F F以"0x" F前綴的小寫十六進位制字串。如果 *integer* 不是 Python *int* 物件，則必須定義一個 `__index__()` method F且回傳一個整數。舉例來F:

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

如果要將整數轉F F大寫或小寫的十六進位制字串，F可選擇有無"0x" 前綴，則可以使用如下方法:

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

可參考*format()* 取得更多資訊。

另請參F*int()* 將十六進位制字串轉F F以 16 F基數的整數。

> **備F**
>
> 如果要取得浮點數的十六進位制字串形式，請使用*float.hex()* method。

id(*object*, /)

回傳物件的" 識F性" 。該值是一個整數，在此物件的生命F期中保證是唯一且F定的。兩個生命期不重F的物件可能具有相同的*id()* 值。

這是該物件在記憶體中的位址。

引發一個附帶引數 `id` 的稽核事件 `builtins.id`。

input()

input(*prompt*, /)

如果有提供 *prompt* 引數，則將其寫入標準輸出，末尾不帶F行符。接下來，該函式從輸入中讀取一行，將其轉F F字串（去除末尾的F行符）F回傳。當讀取到 EOF 時，則引發*EOFError*。例如:

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

如果載入了*readline* module，*input()* 將使用它來提供F雜的行編輯和歷史記F功能。

引發一個附帶讀取輸入前的引數 `prompt` 的稽核事件 `builtins.input`。

引發一個附帶成功讀取結果的稽核事件 `builtins.input/result`。

class int(*number=0*, /)

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

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

例如:

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

如果引數定義了 `__int__()`，則 `int(x)` 會回傳 `x.__int__()`。如果引數定義了 `__index__()` 則回傳 `x.__index__()`。對於浮點數，則會向零的方向無條件舍去。

如果引數不是數字或如果有給定 *base*，則它必須是個字串、*bytes* 或*bytearray* 實例，表示基數 (radix) *base* 中的整數。可選地，字串之前可以有 `+` 或 `-`（中間🄵有空格）、可有個前導的零、也可被空格包圍、或在數字間有單一底🄵。

一個 n 進制的整數字串，包含各個代表 0 到 n-1 的數字，0–9 可以用任何 Unicode 十進制數字表示，10–35 可以用 `a` 到 `z`（或 `A` 到 `Z`）表示。預設的 *base* 是 10。允許的進位制有 0、2–36。2、8、16 進位制的字串可以在程式碼中用 `0b`/`0B`、`0o`/`0O`、`0x`/`0X` 前綴來表示，如同程式碼中的整數文字。進位制🄵 0 的字串將以和程式碼整數字面值 (integer literal in code) 類似的方式來直譯，最後由前綴🄵定的結果會是 2、8、10、16 進制中的一個，所以 `int('010', 0)` 是非法的，但 `int('010')` 和 `int('010', 8)` 是有效的。

整數型🄵定義請參🄵數值型🄵 --- *int*、*float*、*complex*。

在 3.4 版的變更: 如果 *base* 不是*int* 的實例，但 *base* 物件有 `base.__index__` method，則會呼叫該 method 來取得此進位制所需的整數。以前的版本使用 `base.__int__` 而不是 `base.__index__`。

在 3.6 版的變更: 可以使用底🄵將程式碼文字中的數字進行分組。

在 3.7 版的變更: 第一個參數🄵僅限位置參數。

在 3.8 版的變更: 如果未定義 `__int__()` 則會回退到 `__index__()`。

在 3.11 版的變更: *int* 的字串輸入和字串表示法可以被限制，以避免阻斷服務攻擊 (denial of service attack)。在字串 *x* 轉🄵🄵*int* 時已超出限制，或是在*int* 轉🄵🄵字串時將會超出限制時，會引發*ValueError*。請參🄵整數字串轉🄵的長度限制🄵明文件。

在 3.14 版的變更: *int()* 不再委派給 `__trunc__()` method。

isinstance(*object*, *classinfo*, */*)

如果 *object* 引數是 *classinfo* 引數的實例，或者是（直接、間接或*virtual*）subclass 的實例，則回傳 `True`。如果 *object* 不是給定型🄵的物件，函式始終回傳 `False`。如果 *classinfo* 是包含物件型🄵的 tuple（或多個遞🄵 tuple）或一個包含多種型🄵的聯合型🄵 *(Union Type)*，若 *object* 是其中的任何一個物件的實例則回傳 `True`。如果 *classinfo* 既不是型🄵，也不是型🄵 tuple 或型🄵的遞🄵 tuple，那🄵會引發*TypeError* 🄵常。若是先前檢查已經成功，*TypeError* 可能不會再因🄵不合格的型🄵而被引發。

在 3.10 版的變更: *classinfo* 可以是一個聯合型🄵 *(Union Type)*。

issubclass(*class*, *classinfo*, */*)

如果 *class* 是 *classinfo* 的 subclass（直接、間接或*virtual*），則回傳 `True`。*classinfo* 可以是 class 物件的 tuple（或遞🄵地其他類似 tuple）或是一個聯合型🄵 *(Union Type)*，此時若 *class* 是 *classinfo* 中任一元素的 subclass 時則回傳 `True`。其他情🄵，會引發*TypeError*。

在 3.10 版的變更: *classinfo* 可以是一個聯合型🄵 *(Union Type)*。

iter(*iterable*, */*)

iter(*callable*, *sentinel*, */*)

回傳一個*iterator* 物件。根據是否存在第二個引數，第一個引數的意義是非常不同的。如果🄵有第二個引數，該單一引數必須是支援*iterable* 協定（有 `__iter__()` method）的集合物件，或必須支

援序列協定（有 `__getitem__()` 方法，且數字引數從 `0` 開始）。如果它不支援這些協定，會引發 *TypeError*。如果有第二個引數 *sentinel*，那🄵第一引數必須是可呼叫的物件，這種情🄵下生成的 iterator，每次🄵代呼叫 *__next__()* 時會不帶引數地呼叫 *callable*；如果回傳的結果是 *sentinel* 則引發 *StopIteration*，否則回傳呼叫結果。

另請參🄵🄵代器型🄵。

iter() 的第二種形式有一個好用的應用，是能🄵建立一個區塊🄵讀器 (block-reader)。例如，從二進位資料庫檔案中讀取固定寬度的區塊，直到檔案的結尾：

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

len(*object*, /)

回傳物件的長度（元素個數）。引數可以是序列（如 string、bytes、tuple、list 或 range）或集合（如 dictionary、set 或 frozen set）。

如果物件長度大於 *sys.maxsize*，像是 *range(2 ** 100)*，則 `len` 會引發 *OverflowError*。

class list(*iterable=()*, /)

除了是函式，*list* 也是可變序列型🄵，詳情請參🄵*List（串列）* 和*Sequence Types --- list, tuple, range*。

locals()

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

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

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

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

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

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

Calling `locals()` as part of a comprehension in a function, generator, or coroutine is equivalent to calling it in the containing scope, except that the comprehension's initialised iteration variables will be included. In other scopes, it behaves as if the comprehension were running as a nested function.

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

在 3.12 版的變更: The behaviour of `locals()` in a comprehension has been updated as described in **PEP 709**.

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

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

🄵生一個將 *function* 應用於 *iterable* 中所有項目🄵 yield 回傳結果的🄵代器。如果傳遞了額外的 *iterables* 引數，則 *function* 必須接受相同個數的引數，且 *function* 會平行地被應用於所有可🄵代物件的項目。當有多個可🄵代物件時，項目最少的可🄵代物件耗盡時🄵代器也會結束。如果 *strict* 🄵

`True` 且其中一個可🄵代物件在其他可🄵代物件之前耗盡，則會🄵出*ValueError*。如果函式的輸入已經被編排🄵引數的元組，請參🄵*itertools.starmap()*。

在 3.14 版的變更: 增加了 *strict* 參數。

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

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

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

回傳 iterable 中最大的元素，或者回傳兩個以上的引數中最大的。

如果只提供了一個位置引數，它必須是個*iterable*，iterable 中最大的元素會被回傳。如果提供了兩個或以上的位置引數，則回傳最大的位置引數。

這個函式有兩個選擇性的僅限關鍵字引數。*key* 引數能指定單一引數所使用的排序函式，如同*list.sort()* 的使用方式。*default* 引數是當 iterable 🄵空時回傳的物件。如果 iterable 🄵空，🄵且🄵有提供 *default*，則會引發*ValueError*。

如果有多個最大元素，則此函式將回傳第一個找到的。這和其他穩定排序工具如 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 一致。

在 3.4 版的變更: 新增 *default* 僅限關鍵字參數。

在 3.8 版的變更: *key* 可以🄵 `None`。

class memoryview(*object*)

回傳由給定的引數所建立之「memory view（記憶體檢視）」物件。有關詳細資訊，請參🄵*Memory Views*。

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

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

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

回傳 iterable 中最小的元素，或者回傳兩個以上的引數中最小的。

如果只提供了一個位置引數，它必須是*iterable*，iterable 中最小的元素會被回傳。如果提供了兩個以上的位置引數，則回傳最小的位置引數。

這個函式有兩個選擇性的僅限關鍵字引數。*key* 引數能指定單一引數所使用的排序函式，如同*list.sort()* 的使用方式。*default* 引數是當 iterable 🄵空時回傳的物件。如果 iterable 🄵空，🄵且🄵有提供 *default*，則會引發*ValueError*。

如果有多個最小元素，則此函式將回傳第一個找到的。這和其他穩定排序工具如 `sorted(iterable, key=keyfunc)[0]` 和 `heapq.nsmallest(1, iterable, key=keyfunc)` 一致。

在 3.4 版的變更: 新增 *default* 僅限關鍵字參數。

在 3.8 版的變更: *key* 可以🄵 `None`。

next(*iterator*, /)

next(*iterator*, *default*, /)

透過呼叫*iterator* 的*__next__()* method 取得下一個元素。如果 iterator 耗盡，則回傳給定的預設值 *default*，如果🄵有預設值則引發*StopIteration*。

class object

這是所有其他 class 的基礎，它具有所有 Python class 實例的通用 method。當建構函式被呼叫時，它會回傳一個新的🄵有特徵 (featureless) 的物件。這個建構函式不接受任何引數。

> **備🄵**
>
> 由於*object* 實例🄵有 `__dict__` 屬性，因此無法將任意屬性賦給*object* 的實例。

oct(*integer*, */*)

將一個整數轉變🄵一個前綴🄵"0o" 的八進位制字串。回傳結果是一個有效的 Python 運算式。如果 *integer* 不是 Python 的*int* 物件，那它需要定義 `__index__()` method 回傳一個整數。舉例來🄵:

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

如果要將整數轉🄵🄵八進位制字串，不論是否具備"0o" 前綴，都可以使用下面的方法。

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

可參考*format()* 取得更多資訊。

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

開🄵 *file* 🄵回傳對應的檔案物件。如果該檔案不能開🄵，則引發*OSError*。關於使用此函式的更多方法，請參🄵 tut-files。

file 是一個類路徑物件，是將被開🄵之檔案的路徑（🄵對路徑或目前工作目🄵的相對路徑），或是要被包裝 (wrap) 檔案的整數檔案描述器 (file descriptor)。（如果有給定檔案描述器，它會隨著回傳的 I/O 物件關閉而關閉，除非 *closefd* 被設🄵 `False`。）

mode 是一個選擇性字串，用於指定開🄵檔案的模式。預設值是 `'r'`，這意味著它以文字模式開🄵🄵讀取。其他常見模式有：寫入 `'w'`（會舍去已經存在的檔案）、唯一性建立 `'x'`、追加寫入 `'a'`（在一些 Unix 系統上，無論目前的檔案指標在什🄵位置，所有寫入都會追加到檔案末尾）。在文字模式，如果🄵有指定 *encoding*，則根據電腦平臺來🄵定使用的編碼：呼叫*locale.getencoding()* 來取得目前的本地編碼。（要讀取和寫入原始 bytes，請使用二進位制模式且不要指定 *encoding*。）可用的模式有：

字元	意義
`'r'`	讀取（預設）
`'w'`	寫入，會先清除檔案🄵容
`'x'`	唯一性建立，如果文件已存在則會失敗
`'a'`	寫入，如果檔案存在則在其末端附加🄵容
`'b'`	二進制模式
`'t'`	文字模式（預設）
`'+'`	更新（讀取🄵寫入）

預設的模式是 `'r'`（開🄵🄵讀取文字，同 `'rt'`）。`'w+'` 和 `'w+b'` 模式會開🄵🄵清除檔案。`'r+'` 和 `'r+b'` 模式會開🄵且保留檔案🄵容。

如總覽中所述，Python 能區分二進制和文字的 I/O。在二進制模式下開🄵的檔案（*mode* 引數中含有 `'b'`）會將其🄵容以*bytes* 物件回傳，而不進行任何解碼。在文字模式（預設情🄵，或當 *mode* 引數中含有 `'t'`），檔案的🄵容會以*str* 回傳，其位元組已經先被解碼，使用的是取🄵於平台的編碼系統或是給定的 *encoding*。

備🄵

Python 不會使用底層作業系統對於文字檔案的操作概念；所有的處理都是由 Python 獨自完成的，因此能獨立於不同平台。

buffering 是一個選擇性的整數，用於設定緩衝策略。傳入 0 表示關閉緩衝（僅在二進制模式下被允許），1 表示行緩衝（line buffering，僅在文字模式下可用），而 >1 的整數是指示一個大小固定的區塊

緩衝區 (chunk buffer)，其位元組的數量。請注意，此類指定緩衝區大小的方式適用於二進制緩衝 I/O，但是 `TextIOWrapper`（以 `mode='r+'` 開🄵的檔案）會有另一種緩衝方式。若要在 `TextIOWrapper` 中停用緩衝，可考慮使用 *io.TextIOWrapper.reconfigure()* 的 `write_through` 旗標。若未給定 *buffering* 引數，則預設的緩衝策略會運作如下：

- 二進位檔案會以固定大小的區塊進行緩衝；緩衝區的大小🄵 `max(min(blocksize, 8 MiB), DEFAULT_BUFFER_SIZE)`，當裝置區塊大小可用時。在大多數系統上，緩衝區的長度通常🄵 128 KB。
- 「互動式」文字檔（*isatty()* 回傳 `True` 的檔案）會使用列緩衝。其他文字檔則使用上述的二進制檔案緩衝策略。

encoding 是用於解碼或編碼檔案的編碼系統之名稱。它只應該在文字模式下使用。預設的編碼系統會取🄵於平台（根據 *locale.getencoding()* 回傳的🄵容），但 Python 支援的任何 *text encoding*（文字編碼）都是可以使用的。關於支援的編碼系統清單，請參🄵 *codecs* module。

errors 是一個選擇性的字串，用於指定要如何處理編碼和解碼的錯誤——它不能在二進制模式下使用。有許多不同的標準錯誤處理程式（error handler，在 *Error Handlers* 有列出清單），不過任何已🄵🄵到 *codecs.register_error()* 的錯誤處理程式名稱也都是有效的。標準的名稱包括：

- `'strict'` 如果發生編碼錯誤，則引發 *ValueError* 例外。預設值 `None` 也有相同的效果。
- `'ignore'` 忽略錯誤。請注意，忽略編碼錯誤可能導致資料遺失。
- `'replace'` 會在格式不正確的資料位置插入一個替🄵標🄵（像是 `'?'`）。
- `'surrogateescape'` 會將任何不正確的位元組表示🄵低位代理碼元 (low surrogate code unit)，範圍從 U+DC80 到 U+DCFF。在寫入資料時，這些代理碼元將會被還原回 `surrogateescape` 錯誤處理程式當時所處理的那些相同位元組。這對於處理未知編碼方式的檔案會很好用。
- `'xmlcharrefreplace'` 僅在寫入檔案時可支援。編碼系統不支援的字元會被替🄵🄵適當的 XML 字元參考 (character reference) `&#nnn;`。
- `'backslashreplace'` 會用 Python 的反斜🄵跳🄵序列 (backslashed escape sequence) 替🄵格式不正確的資料。
- `'namereplace'`（也僅在寫入時支援）會將不支援的字元替🄵🄵 `\N{...}` 跳🄵序列。

newline 會🄵定如何剖析資料串流 (stream) 中的🄵行字元。它可以是 `None`、`''`、`'\n'`、`'\r'` 或 `'\r\n'`。它的運作規則如下：

- 從資料串流讀取輸入時，如果 *newline* 是 `None`，則會🄵用通用🄵行模式。輸入資料中的行結尾可以是 `'\n'`、`'\r'` 或 `'\r\n'`，這些符號會被轉🄵🄵 `'\n'` 之後再回傳給呼叫方。如果是 `''`，也會🄵用通用🄵行模式，但在回傳給呼叫方時，行尾符號不會被轉🄵。如果它是任何其他有效的值，則輸入資料的行只會由給定的字串做結尾，且在回傳給呼叫方時，行尾符號不會被轉🄵。
- 將輸出寫入資料串流時，如果 *newline* 是 `None`，則被寫入的任何 `'\n'` 字元都會轉🄵🄵系統預設的行分隔符號 *os.linesep*。如果 *newline* 是 `''` 或 `'\n'`，則不做任何轉🄵。如果 *newline* 是任何其他有效的值，則寫入的任何 `'\n'` 字元都將轉🄵🄵給定的字串。

如果 *closefd* 是 `False`，且給定的 *file* 引數是一個檔案描述器而不是檔名，則當檔案關閉時，底層的檔案描述器會保持開🄵狀態。如果有給定一個檔名，則 *closefd* 必須是 `True`（預設值）；否則將引發錯誤。

透過以 *opener* 傳遞一個可呼叫物件，就可以自訂開🄵函式。然後透過以引數 (*file*, *flags*) 呼叫 *opener*，就能取得檔案物件的底層檔案描述器。*opener* 必須回傳一個開🄵的檔案描述器（將 *os.open* 作🄵 *opener* 傳入，在功能上的結果會相當於傳入 `None`）。

新建立的檔案是不可繼承的。

下面的範例使用 *os.open()* 函式回傳值當作 *dir_fd* 的參數，從給定的目🄵中用相對路徑開🄵檔案：

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
```

(繼續下一頁)

(繼續上一頁)

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

open() 函式回傳的*file object* 型🄵取🄵於模式。當*open()* 是在文字模式中開🄵檔案時（'w'、'r'、'wt'、'rt' 等），它會回傳*io.TextIOBase* 的一個 subclass（具體來🄵，就是*io.TextIOWrapper*）。使用有緩衝的二進制模式開🄵檔案時，回傳的 class 則會是*io.BufferedIOBase* 的 subclass。確切的 class 各不相同：在讀取的二進制模式，它會回傳*io.BufferedReader*；在寫入和附加的二進制模式，它會回傳*io.BufferedWriter*，而在讀／寫模式，它會回傳*io.BufferedRandom*。當緩衝被停用時，會回傳原始資料串流*io.FileIO*，它是*io.RawIOBase* 的一個 subclass。

另請參🄵檔案操作模組，例如*fileinput*、*io*（定義了*open()* 的 module）、*os*、*os.path*、*tempfile* 以及*shutil*。

引發一個附帶引數 path、mode、flags 的稽核事件 open。

mode 和 flags 引數可能會被原始的呼叫所修改或推論 (infer)。

在 3.3 版的變更:

- 增加了 *opener* 參數。
- 增加了 'x' 模式。
- 過去引發的*IOError*，現在是*OSError* 的🄵名。
- 如果檔案已存在但使用了唯一性建立模式 ('x')，現在會引發*FileExistsError*。

在 3.4 版的變更:

- 檔案在此版本開始是不可繼承的。

在 3.5 版的變更:

- 如果系統呼叫被中斷，但訊號處理程式🄵有引發例外，此函式現在會重試系統呼叫，而不是引發*InterruptedError* 例外（原因詳見 **PEP 475**）。
- 增加了 'namereplace' 錯誤處理程式。

在 3.6 版的變更:

- 增加對於實作*os.PathLike* 物件的支援。
- 在 Windows 上，開🄵一個控制臺緩衝區可能會回傳*io.RawIOBase* 的 subclass，而不是*io.FileIO*。

在 3.11 版的變更: 'U' 模式被移除。

ord(*character*, /)

Return the ordinal value of a character.

如果引數是單字元字串，則回傳該字元的 Unicode 編碼位置。例如 ord('a') 回傳整數 97、ord('€')（歐元符號）回傳 8364。這是*chr()* 的逆函式。

If the argument is a *bytes* or *bytearray* object of length 1, return its single byte value. For example, ord(b'a') returns the integer 97.

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

回傳 *base* 的 *exp* 次方；如果 *mod* 存在，則回傳 *base* 的 *exp* 次方對 *mod* 取余數（比直接呼叫 pow(base, exp) % mod 計算更高效）。兩個引數形式的 pow(exp, exp) 等價於次方運算子: base**exp。

引數必須🄵數值型🄵。當運算元型🄵🄵混合型🄵時，會套用二元算術運算子的🄵制轉型 (coercion) 規則。對於*int* 運算元，運算結果會（在🄵制轉型後）與運算元的型🄵相同，除非第二個引數是負數；在這種情🄵下，所有的引數都會被轉🄵🄵浮點數🄵得到浮點數的結果。例如，pow(10, 2) 會回傳 100，但 pow(10, -2) 會回傳 0.01。如果底數 (base) 是型🄵🄵*int* 或*float* 的負數且指數 (exponent) 🄵非整數，則會傳回🄵數結果。例如 pow(-9, 0.5) 會回傳一個接近 3j 的值。然而，如

果底數 (base) 是型🄵🄵*int* 或*float* 的負數且指數🄵整數，則會傳回浮點數結果。例如 `pow(-9, 2.0)` 會回傳 `81.0`。

對於*int* 運算元 *base* 和 *exp*，如果有給定 *mod*，則 *mod* 也必須是整數型🄵，且 *mod* 必須不🄵零。如果有給定 *mod* 且 *exp* 🄵負，則 *base* 必須與 *mod* 互質。在這種情🄵下，會回傳 `pow(inv_base, -exp, mod)`，其中 *inv_base* 是 *base* 對 *mod* 的模倒數 (inverse modulo)。

以下是一個計算 `38` 對 `97` 取模倒數的範例：

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

在 3.8 版的變更: 對於*int* 運算元，現在 `pow` 的三引數形式允許第二個引數🄵負數，也容許模倒數的計算。

在 3.8 版的變更: 允許關鍵字引數。在此之前只支援位置引數。

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

將 *objects* 列印到文字資料串流 *file*，用 *sep* 分隔🄵以 *end* 結尾。如果有給定 *sep*、*end*、*file* 和 *flush*，那🄵它們必須是關鍵字引數的形式。

所有的非關鍵字引數都會像是*str()* 操作一樣地被轉🄵🄵字串，🄵寫入資料串流，彼此以 *sep* 分隔，🄵以 *end* 結尾。*sep* 和 *end* 都必須是字串；它們也可以是 `None`，這表示使用預設值。如果🄵有給定 *objects*，*print()* 就只會寫入 *end*。

file 引數必須是一個有 `write(string)` method 的物件；如果🄵有給定或被設🄵 `None`，則將使用*sys.stdout*。因🄵要列印的引數會被轉🄵🄵文字字串，所以*print()* 不能用於二進位模式的檔案物件。對於此類物件，請改用 `file.write(...)`。

輸出緩衝通常會由 *file* 🄵定。但是如果 *flush* 🄵 true，則資料串流會被🄵制清除。

在 3.3 版的變更: 增加了 *flush* 關鍵字引數。

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

回傳 property 屬性。

fget 是一個用於取得屬性值的函式，*fset* 是一個用於設定屬性值的函式，*fdel* 是一個用於🄵除屬性值的函式，而 *doc* 會🄵該屬性建立一個🄵明字串。

一個典型的用途是定義一個受管理的屬性 `x`：

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

    def getx(self):
        return self._x

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

    def delx(self):
        del self._x

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

如果 *c* 是 *C* 的一個實例，則 `c.x` 將會叫用取得器 (getter)，`c.x = value` 會呼叫設定器 (setter)，而 `del c.x` 會叫用🄵除器 (deleter)。

如果有給定 *doc*，它將會是 property 屬性的🄵明字串。否則，property 會🄵🄵 *fget* 的🄵明字串（如果它存在的話）。這樣一來，就能🄵輕松地使用*property()* 作🄵裝飾器來建立唯讀屬性：

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

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

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

@getter

@setter

@deleter

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

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

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

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

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

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

在 3.5 版的變更: The docstrings of property objects are now writeable.

__name__

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

在 3.13 版被加入.

class range(*stop*, */*)

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

Rather than being a function, *range* is actually an immutable sequence type, as documented in *Ranges* and *Sequence Types --- list, tuple, range*.

repr(*object*, */*)

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

This class has a custom representation that can be evaluated:

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

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

reversed(*object*, /)

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

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

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting *round()*, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

For a general Python object `number`, `round` delegates to `number.__round__`.

> **備🄵**
>
> The behavior of *round()* for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See tut-fp-issues for more information.

class set(*iterable=()*, /)

Return a new *set* object, optionally with elements taken from *iterable*. `set` is a built-in class. See *set* and *Set Types --- set, frozenset* for documentation about this class.

For other containers see the built-in *frozenset*, *list*, *tuple*, and *dict* classes, as well as the *collections* module.

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

This is the counterpart of *getattr()*. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in identifiers unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through *getattr()* etc..

> **備🄵**
>
> Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with *setattr()*.

class slice(*stop*, /)

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

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

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

start

stop

step

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

在 3.12 版的變更: Slice objects are now *hashable* (provided *start*, *stop*, and *step* are hashable).

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

Return a new sorted list from the items in *iterable*.

有兩個選擇性引數，只能使用關鍵字引數來指定。

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

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use *functools.cmp_to_key()* to convert an old-style *cmp* function to a *key* function.

The built-in *sorted()* function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal --- this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

The sort algorithm uses only < comparisons between items. While defining an `__lt__()` method will suffice for sorting, **PEP 8** recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as *max()* that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call reflected the `__gt__()` method.

For sorting examples and a brief sorting tutorial, see sortinghowto.

@staticmethod

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

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

`@staticmethod` 語法是一個函式*decorator* - 參見 function 中的詳細介紹。

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

Static methods in Python are similar to those found in Java or C++. Also, see *classmethod()* for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
def regular_function():
    ...

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

關於 static method 的更多資訊，請參考 types。

在 3.10 版的變更: Static method 現在繼承了 method 屬性（`__module__`、`__name__`、`__qualname__`、`__doc__` 和 `__annotations__`），🄵擁有一個新的 `__wrapped__` 屬性，且🄵如一般函式的可呼叫物件。

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

class str(*object*)

class str(*object*, *encoding*, *errors='strict'*)

class str(*object*, *, *errors*)

Return a *str* version of *object*. See *str()* for details.

`str` is the built-in string *class*. For general information about strings, see *Text Sequence Type --- str*.

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

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

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

在 3.8 版的變更: *start* 參數可被指定🄵關鍵字引數。

在 3.12 版的變更: Summation of floats switched to an algorithm that gives higher accuracy and better commutativity on most builds.

在 3.14 版的變更: Added specialization for summation of complexes, using same algorithm as for summation of floats.

class super

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

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class.

The *object_or_type* determines the *method resolution order* to be searched. The search starts from the class right after the *type*.

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

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

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

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

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement ”diamond diagrams” where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

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

In addition to method lookups, *super()* also works for attribute lookups. One possible use case for this is calling *descriptors* in a parent or sibling class.

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

Also note that, aside from the zero argument form, *super()* is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using *super()*, see guide to using super().

在 3.14 版的變更: *super* objects are now *pickleable* and *copyable*.

class tuple(*iterable=()*, */*)

Rather than being a function, *tuple* is actually an immutable sequence type, as documented in *Tuples* and *Sequence Types --- list, tuple, range*.

class type(*object*, */*)

class type(*name*, *bases*, *dict*, */*, ***kwargs*)

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

The *isinstance()* built-in function is recommended for testing the type of an object, because it takes subclasses into account.

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

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

See also:

- Documentation on attributes and methods on classes.
- *Type Objects*

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

另請參🄵 class-customization。

在 3.6 版的變更: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()

vars(*object*, /)

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

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

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

A *TypeError* exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).

在 3.13 版的變更: The result of calling this function without an argument has been updated as described for the *locals()* builtin.

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

Iterate over several iterables in parallel, producing tuples with an item from each one.

例如:

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

More formally: *zip()* returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument iterables.

Another way to think of *zip()* is that it turns rows into columns, and columns into rows. This is similar to transposing a matrix.

zip() is lazy: The elements won't be processed until the iterable is iterated on, e.g. by a `for` loop or by wrapping in a *list*.

One thing to consider is that the iterables passed to *zip()* could have different lengths; sometimes by design, and sometimes because of a bug in the code that prepared these iterables. Python offers three different approaches to dealing with this issue:

- By default, *zip()* stops when the shortest iterable is exhausted. It will ignore the remaining items in the longer iterables, cutting off the result to the length of the shortest iterable:

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

- *zip()* is often used in cases where the iterables are assumed to be of equal length. In such cases, it's recommended to use the `strict=True` option. Its output is the same as regular *zip()*:

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

  Unlike the default behavior, it raises a *ValueError* if one iterable is exhausted before the others:

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

Without the `strict=True` argument, any bug that results in iterables of different lengths will be silenced, possibly manifesting as a hard-to-find bug in another part of the program.

- Shorter iterables can be padded with a constant value to make all the iterables have the same length. This is done by *itertools.zip_longest()*.

Edge cases: With a single iterable argument, *zip()* returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

Tips and tricks:

- The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n, strict=True)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.
- *zip()* in conjunction with the `*` operator can be used to unzip a list:

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

在 3.10 版的變更: 增加了 `strict` 引數。

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

> **備𝔽**
>
> This is an advanced function that is not needed in everyday Python programming, unlike *importlib.import_module()*.

This function is invoked by the `import` statement. It can be replaced (by importing the *builtins* module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of *__import__()* is also discouraged in favor of *importlib.import_module()*.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. `0` (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling *__import__()* (see **PEP 328** for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

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

The statement `import spam.ham` results in this call:

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

Note how *__import__()* returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

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

Here, the `spam.ham` module is returned from *__import__()*. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use *importlib.import_module()*.

在 3.3 版的變更: Negative values for *level* are no longer supported (which also changes the default value to 0).

在 3.9 版的變更: When the command line options `-E` or `-I` are being used, the environment variable `PYTHONCASEOK` is now ignored.

🄵解

CHAPTER 3

🄵建常數

有一小部分的常數存在於🄵建命名空間中。他們是：

False

在*bool* 型🄵中的 false 值。對於 `False` 的賦值是不合法的，🄵且會🄵出*SyntaxError*。

True

在*bool* 型🄵中的 true 值。對於 `True` 的賦值是不合法的，🄵且會🄵出*SyntaxError*。

None

型🄵 `NoneType` 的唯一值。`None` 經常被使用來表達缺少值，例如未傳送預設的引數至函式時，相對應參數即會被賦予 `None`。對於 `None` 的賦值是不合法的，🄵且會🄵出*SyntaxError*。`None` 是型🄵*NoneType* 的唯一實例。

NotImplemented

會被二元特殊方法 (binary special methods)（如：`__eq__()`、`__lt__()`、`__add__()`、`__rsub__()` 等）所回傳的特殊值，代表著該運算🄵有針對其他型🄵的實作。同理也可以被原地二元特殊方法 (in-place binary special methods)（如：`__imul__()`、`__iand__()` 等）回傳。它不應該被作🄵 boolean（布林）來解讀。`NotImplemented` 是型🄵*types.NotImplementedType* 的唯一實例。

> **備🄵**
>
> 當一個二元 (binary) 或原地 (in-place) 方法回傳 `NotImplemented`，直譯器會嘗試反映該操作到其他型🄵（或是其他後備 (fallback)，取🄵於是哪種運算子）。如果所有的常識都回傳 `NotImplemented`，直譯器會🄵出適當的例外。不正確的回傳 `NotImplemented` 會造成誤導的錯誤訊息或是 `NotImplemented` 值被傳回到 Python 程式碼中。
>
> 請參見實作算術操作 以找到更多範例。

> **警示**
>
> `NotImplemented` 與 `NotImplementedError` 🄵不一樣且不可互🄵。這個常數只能用於上述的情🄵；欲知更多如何正確使用它們的細節，請參見*NotImplementedError*。

在 3.9 版的變更: 在 boolean（布林）上下文中對 `NotImplemented` 求值的操作已被🄵用。

在 3.14 版的變更: 在 boolean 上下文中解讀 `NotImplemented` 現在會引發 *TypeError*。過去自 Python 3.9 以來，這會被解讀成 *True*，🄵發出 *DeprecationWarning*

Ellipsis

與🄵節號"..." 字面相同，是一個經常被用來表示某些東西被省略的物件。對於 `Ellipsis` 的賦值是可能的，但對於 `...` 的賦值會引發 *SyntaxError*。`Ellipsis` 是型🄵 *types.EllipsisType* 的唯一實例。

__debug__

如果 Python 🄵有被以 `-O` 選項🄵動，則此常數🄵 true。請參見 `assert` 陳述式。

備🄵

None、*False*、*True*，以及 *__debug__* 都是不能被重新賦值的（任何對它們的賦值，即使是屬性的名稱，也會🄵出 *SyntaxError*）。因此，它們可以被視🄵" 真正的" 常數。

3.1 由 `site` module（模組）所添增的常數

site module（模組）（在🄵動期間自動 import，除非有給予 `-S` 指令行選項）會添增一些常數到🄵建命名空間 (built-in namespace) 中。它們在互動式直譯器中是很有幫助的，但不應該在程式 (programs) 中被使用。

quit(*code=None*)

exit(*code=None*)

當印出物件時，會印出一個訊息："Use quit() or Ctrl-D (i.e. EOF) to exit"。當直接在互動式直譯器中存取或作🄵函式呼叫時，會引發 *SystemExit* 🄵帶有指定的返回碼（exit code）。

help

當印出此物件時，會印出訊息"Type help() for interactive help, or help(object) for help about object."，🄵在互動式直譯器中直接存取時，會叫用🄵建的幫助系統（參見 *help()*）。

copyright

credits

當印出或是呼叫此物件時，分🄵會印出版權與致謝的文字。

license

當印出此物件時，會印出訊息"Type license() to see the full license text"，🄵在呼叫時以分頁形式印出完整的許可證文字（一次一整個畫面）。

CHAPTER 4

🄵建型🄵

以下章節描述了直譯器中🄵建的標准型🄵。

主要🄵建型🄵🄵數字、序列、對映、class（類🄵）、實例和例外。

有些集合類🄵是 mutable（可變的）。那些用於原地 (in-place) 加入、移除或重新排列其成員且不回傳特定項的 method（方法），也只會回傳 None 而非集合實例自己。

某些操作已被多種物件型🄵支援；特🄵是實務上所有物件都已經可以做相等性比較、真值檢測及被轉🄵🄵字串（使用*repr()* 函式或稍有差🄵的*str()* 函式），後者🄵當物件傳入*print()* 函式印出時在背後被呼叫的函式。

4.1 真值檢測

任何物件都可以進行檢測以判斷是否🄵真值，以便在 if 或 while 條件中使用，或是作🄵如下所述 boolean（布林）運算之運算元所用。

預設情🄵下，一個物件會被視🄵真值，除非它的 class 定義了會回傳 False 的 __bool__() method 或是定義了會回傳零的 __len__() method。[1] 以下列出了大部分會被視🄵 false 的🄵建物件：

- 定義🄵 false 之常數：None 與 False
- 任何數值型🄵的零：0、0.0、0j、Decimal(0)、Fraction(0, 1)
- 空的序列和集合：''、()、[]、{}、set()、range(0)

除非另有特🄵🄵明，🄵生 boolean 結果的操作或🄵建函式都會回傳 0 或 False 作🄵假值、1 或 True 作🄵真值。（重要例外：boolean 運算 or 和 and 回傳的是其中一個運算元。）

4.2 Boolean（布林）運算 --- and, or, not

下方🄵 Boolean 運算，按優先順序排序：

運算	結果	🄵解
x or y	假如 *x* 🄵真，則 *x*，否則 *y*	(1)
x and y	假如 *x* 🄵假，則 *x*，否則 *y*	(2)
not x	假如 *x* 🄵假，則 True，否則 False	(3)

[1] Additional information on these special methods may be found in the Python Reference Manual (customization).

□解：

(1) 這是一個短路運算子，所以他只有在第一個引數□假時，才會對第二個引數求值。

(2) 這是一個短路運算子，所以他只有在第一個引數□真時，才會對第二個引數求值。

(3) `not` 比非 Boolean 運算子有較低的優先權，因此 `not a == b` 可直譯□ `not (a == b)`，而 `a == not b` 會導致語法錯誤。

4.3 比較運算

在 Python □共有 8 種比較運算。他們的優先順序都相同（皆優先於 Boolean 運算）。比較運算可以任意的串連；例如，`x < y <= z` 等同於 `x < y and y <= z`，差□只在於前者的 *y* 只有被求值一次（但在這兩個例子中，當 `x < y` □假時，*z* 皆不會被求值）。

這個表格統整所有比較運算：

運算	含義
`<`	小於
`<=`	小於等於
`>`	大於
`>=`	大於等於
`==`	等於
`!=`	不等於
`is`	物件識□性
`is not`	否定的物件識□性

除了不同的數值型□外，不同型□的物件不能進行相等比較。運算子 `==` 總有定義，但在某些物件型□（例如，class 物件）時，運算子會等同於 `is`。其他運算子 `<`、`<=`、`>` 及 `>=` 皆僅在有意義的部分有所定義；例如，當其中一個引數□□數時，將引發一個 *TypeError* 的例外。

一個 class 的非相同實例通常會比較□不相等，除非 class 有定義 `__eq__()` method。

一個 class 的實例不可以與其他相同 class 的實例或其他物件型□進行排序，除非 class 定義足□的 method，包含 `__lt__()`、`__le__()`、`__gt__()` 及 `__ge__()`（一般來□，使用 `__lt__()` 及 `__eq__()` 就可以滿足常規意義上的比較運算子）。

無法自定義 `is` 與 `is not` 運算子的行□；這兩個運算子也可以運用在任意兩個物件且不會引發例外。

此外，擁有相同的語法優先序的 `in` 及 `not in` 兩種運算皆被可□代物件或者有實作 `__contains__()` method 的型□所支援。

4.4 數值型□ --- `int`、`float`、`complex`

數值型□共有三種：整數、浮點數及 □數。此外，Boolean □整數中的一個子型□。整數有無限的精度。浮點數通常使用 C □面的 `double` 實作。關於在你程式所運作的機器上之浮點數的精度及□部表示法可以在 *sys.float_info* 進行查找。□數包含實數及□數的部分，這兩部分各自是一個浮點數。若要從一個□數 *z* 提取這兩部分，需使用 `z.real` 及 `z.imag`。（標準函式庫包含額外的數值型□，像是 *fractions.Fraction* 表示有理數，而 *decimal.Decimal* 表示可由使用者制定精度的浮點數。）

數字是由字面數值或□建公式及運算子的結果所□生的。未經修飾的字面數值（含十六進位、八進位及二進位數值）會 yield 整數。包含小數點或指數符號的字面數值會 yield 浮點數。在數值後面加上 `'j'` 或是 `'J'` 會 yield 一個□數（意即一個實數□ 0 的□數）。你也可以將整數與浮點數相加以得到一個有實部與□部的□數。

建構函式：*int()*、*float()* 及 *complex()* 可以用來□生特定型□的數字。

Python 完全支援混和運算：當一個二元運算子的運算元有不同數值型□時，「較窄」型□的運算元會被拓寬到另一個型□的運算元；在此處，整數窄於浮點數。□數及實數運算元的運算是由一般的數學公式所定義，例如：

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

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

所有數值型🄵（除🄵數外）皆支援以下的運算（有關運算的先後順序，詳見 operator-summary）：

運算	結果	🄵解	完整文件
`x + y`	*x* 及 *y* 的加總		
`x - y`	*x* 及 *y* 的相🄵		
`x * y`	*x* 及 *y* 的相乘		
`x / y`	*x* 及 *y* 相除之商		
`x // y`	*x* 及 *y* 的整數除法	(1)(2)	
`x % y`	`x / y` 的余數	(2)	
`-x`	*x* 的負數		
`+x`	*x* 不變		
`abs(x)`	*x* 的🄵對值或量 (magnitude)		*abs()*
`int(x)`	將 *x* 轉🄵整數	(3)(6)	*int()*
`float(x)`	將 *x* 轉🄵浮點數	(4)(6)	*float()*
`complex(re, im)`	一個🄵數，其實部🄵 *re*，🄵部🄵 *im*。*im* 預設🄵零。	(6)	*complex()*
`c.conjugate()`	🄵🄵數 *c* 的共🄵🄵數		
`divmod(x, y)`	一對 `(x // y, x % y)`	(2)	*divmod()*
`pow(x, y)`	*x* 的 *y* 次方	(5)	*pow()*
`x ** y`	*x* 的 *y* 次方	(5)	

🄵解：

(1) 也被稱🄵整數除法。對於型🄵🄵*int* 的運算元來🄵，結果之型🄵會是*int*。對於型🄵🄵*float* 的運算元來🄵，結果之型🄵會是*float*。一般來🄵，結果會是一個整數，但其型🄵不一定會是*int*。結果總是會往負無窮大的方向取整數值：`1//2` 🄵 `0`、`(-1)//2` 🄵 `-1`、`1//(-2)` 🄵 `-1` 及 `(-1)//(-2)` 🄵 `0`。

(2) 不可用於🄵數。在適當情形下，可使用*abs()* 轉🄵🄵浮點數。

(3) 從*float* 轉🄵🄵*int* 會導致截斷🄵排除小數部分。詳見*math.floor()* 及*math.ceil()* 以了解更多轉🄵方式。

(4) 浮點數也接受帶有可選的前綴"+" 及"-" 的"nan" 及"inf" 字串，其分🄵代表非數字（NaN）及正負無窮。

(5) Python 將 `pow(0, 0)` 及 `0 ** 0` 定義🄵 `1` 這是程式語言的普遍做法。

(6) 字面數值接受包含數字 `0` 到 `9` 或任何等效的 Unicode 字元（具有 `Nd` 屬性的 code points（編碼位置））。

請參🄵 Unicode 標准以了解具有 `Nd` 屬性的 code points 完整列表。

所有*numbers.Real* 型🄵（*int* 及*float*）也適用下列運算：

運算	結果
math.trunc(x)	*x* 截斷🄵*Integral*
round(x[, n])	*x* 進位至小數點後第 *n* 位，使用偶數舍入法。若省略 *n*，則預設🄵 0。
math.floor(x)	小於等於 *x* 的最大*Integral*
math.ceil(x)	大於等於 *x* 的最小*Integral*

關於其他數值運算請詳見*math* 及*cmath* modules（模組）。

[2] As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

4.4.1 整數型🄵的位元運算

位元運算只對整數有意義。位元運算的計算結果就如同對二的補數執行無窮多個符號位元。

二元位元運算的優先順序皆低於數字運算，但高於比較運算；一元運算 ~ 與其他一元數值運算有一致的優先順序（+ 及 -）。

這個表格列出所有位元運算🄵以優先順序由先至後排序。

運算	結果	🄵解
`x | y`	*x* 及 *y* 的位元 或	(4)
`x ^ y`	*x* 及 *y* 的位元 邏輯互斥或	(4)
`x & y`	*x* 及 *y* 的位元 與	(4)
`x << n`	*x* 往左移動 *n* 個位元	(1)(2)
`x >> n`	*x* 往右移動 *n* 個位元	(1)(3)
`~x`	反轉 *x* 的位元	

🄵解：

(1) 負數位移是不被允許🄵會引發 *ValueError* 的錯誤。

(2) 向左移動 *n* 個位元等同於乘以 `pow(2, n)`。

(3) 向右移動 *n* 個位元等同於向下除法除以 `pow(2, n)`。

(4) 在有限的二的補數表示法中執行這些計算（一個有效位元寬度🄵 `1 + max(x.bit_length(), y.bit_length())` 或以上）🄵至少有一個額外的符號擴展位元，便足以得到與無窮多個符號位元相同的結果。

4.4.2 整數型🄵的附加方法

整數型🄵實作了 *numbers.Integral 抽象基底類*🄵。此外，它提供了一些方法：

`int.`**`bit_length`**`()`

回傳以二進位表示一個整數所需要的位元數，不包括符號及首位的零：

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

更准確來🄵，若 `x` 非🄵零，則 `x.bit_length()` 會得出滿足 `2**(k-1) <= abs(x) < 2**k` 的單一正整數 `k`。同樣地，當 `abs(x)` 足🄵小到能正確地取得舍入的對數，則 `k = 1 + int(log(abs(x), 2))`。若 `x` 🄵零，則 `x.bit_length()` 會回傳 `0`。

等同於：

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

在 3.1 版被加入.

`int.`**`bit_count`**`()`

回傳在🄵對值表示的二進位中 1 的個數。這也被稱作母體計數。舉例來🄵：

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

(繼續下一頁)

(繼續上一頁)

```
3
>>> (-n).bit_count()
3
```

等同於:

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

在 3.10 版被加入.

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

回傳表示一個整數的一列位元組。

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

此整數會使用 *length* 位元組表示，🄵且預設🄵 1。如果該整數無法用給定的位元組數來表示，則會引發*OverflowError*。

byteorder 引數🄵定了用來表示整數的位元組順序🄵且預設🄵 `"big"`。如果 byteorder 是 `"big"`，最重要的位元組位於位元組陣列的開頭。如果 byteorder 是 `"little"`，最重要的位元組位於位元組陣列的結尾。

signed 引數🄵定是否使用二的補數來表示整數。如果 *signed* 是 `False` 🄵且給定了一個負整數，則會引發*OverflowError*。*signed* 的預設值是 `False`。

預設值可以方便地將一個整數轉🄵🄵單一位元組物件:

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

然而，使用預設引數時，不要嘗試轉🄵大於 255 的值，否則你將會得到一個*OverflowError*。

等同於:

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

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

在 3.2 版被加入.

在 3.11 版的變更: 🄵 `length` 和 `byteorder` 添加了預設引數值。

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

回傳由給定的位元組陣列表示的整數。

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
```

(繼續下一頁)

(繼續上一頁)

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

引數 *bytes* 必須是一個類位元組物件或是一個🄴生位元組的可🄴代物件。

byteorder 引數🄴定了用來表示整數的位元組順序🄴且預設🄴 `"big"`。如果 *byteorder* 是 `"big"`，最重要的位元組位於位元組陣列的開頭。如果 *byteorder* 是 `"little"`，最重要的位元組位於位元組陣列的結尾。若要請求主機系統的本機位元組順序，請使用 *sys.byteorder* 作🄴位元組順序值。

signed 引數指示是否使用二的補數來表示整數。

等同於：

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

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

    return n
```

在 3.2 版被加入.

在 3.11 版的變更: 🄵 `byteorder` 添加了預設引數值。

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

回傳一對整數，其比率等於原始整數🄴且有一個正分母。整數（整個數值）的整數比率總是整數作🄵分子，🄵且 `1` 作🄵分母。

在 3.8 版被加入.

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

回傳 `True`。🄵了與 *float.is_integer()* 的鴨子型🄴相容而存在。

在 3.12 版被加入.

4.4.3 浮點數的附加方法

浮點數型🄴實作了 *numbers.Real* 抽象基底類🄴。浮點數也有下列附加方法。

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

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

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

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

在 3.14 版被加入.

float.**as_integer_ratio**()

回傳一對整數，其比率完全等於原始浮點數。比率是在最低條件下🄵且有一個正分母。在無窮大時引發*OverflowError*，在 NaN 時引發*ValueError*。

float.**is_integer**()

如果浮點數實例是有限的🄵且具有整數值，則回傳 True，否則回傳 False：

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

兩個 methods 皆支援十六進位字串之間的轉🄵。由於 Python 的浮點數🄵部以二進位數值儲存，將浮點數轉🄵🄵或從 十進位字串通常涉及一個小的四舍五入誤差。相反地，十六進位字串允許精確表示和指定浮點數。這在除錯和數值工作中可能會有用。

float.**hex**()

回傳浮點數的十六進位字串表示。對於有限浮點數，此表示方式總是包含一個前導 0x 及一個尾部 p 和指數。

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

Class method 回傳由十六進位字串 *s* 表示的浮點數。字串 *s* 可能有前導及尾部的空白。

請注意*float.hex()* 是一個實例 method，而*float.fromhex()* 是一個 class method。

一個十六進位字串的形式如下：

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

其中可選的 sign 可以是 + 或 -，integer 和 fraction 是十六進位數字的字串，而 exponent 是一個十進位整數🄵且有一個可選的前導符號。大小寫不重要，🄵且整數或小數部分至少有一個十六進位數字。這個語法與 C99 標準的第 6.4.4.2 節指定的語法相似，也與 Java 1.5 以後的語法相似。特🄵是*float.hex()* 的輸出可用作 C 或 Java 程式碼中的十六進位浮點數文字，🄵且 C 的 %a 格式字元或 Java 的 Double.toHexString 🄵生的十六進位字串可被*float.fromhex()* 接受。

請注意指數是以十進位而非十六進位寫入，🄵且它給出了乘以🄵數的 2 的次方。例如，十六進位字串 0x3.a7p10 表示浮點數 (3 + 10./16 + 7./16**2) * 2.0**10，或 3740.0：

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

對 3740.0 應用反向轉🄵會給出一個不同的十六進位字串，它表示相同的數字：

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

4.4.4 🄵數的附加方法

complex 型🄵實作了*numbers.Complex 抽象基底類*🄵。complex 也有下列附加方法。

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

將一個數字轉🄵🄵🄵數的類🄵方法。

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

在 3.14 版被加入.

4.4.5 數值型𠀀的雜𠀀

對於數字 x 和 y，可能是不同型𠀀，當 x == y 時，hash(x) == hash(y) 是一個要求（詳見 __hash__() method 的文件以獲得更多細節）。𠀀了實作的便利性和效率跨越各種數值型𠀀（包括*int*、*float*、*decimal.Decimal* 和*fractions.Fraction*）Python 的數值型𠀀的雜𠀀是基於一個數學函式，它對於任何有理數都是定義的，因此適用於所有*int* 和*fractions.Fraction* 的實例，以及所有有限的*float* 和*decimal.Decimal* 的實例。基本上，這個函式是由簡化的 modulo（模數）P 給出的一個固定的質數 P。P 的值作𠀀*sys.hash_info* 的*modulus* 屬性提供給 Python。

目前在具有 32 位元 C longs 的機器上所使用的質數是 P = 2**31 - 1，而在具有 64 位元 C longs 的機器上𠀀 P = 2**61 - 1 。

以下是詳細的規則：

- 如果 x = m / n 是一個非負的有理數，𠀀且 n 不可被 P 整除，則將 hash(x) 定義𠀀 m * invmod(n, P) % P。其中 invmod(n, P) 𠀀 n 對模數 P 的倒數。
- 如果 x = m / n 是一個非負的有理數，𠀀且 n 可被 P 整除（但 m 不行），則 n 𠀀有 inverse modulo（模倒數）P，𠀀且不適用於上述規則；在這種情𠀀下，將 hash(x) 定義𠀀常數值 sys.hash_info.inf。
- 如果 x = m / n 是一個負的有理數，則將 hash(x) 定義𠀀 -hash(-x)。如果結果的雜𠀀是 -1，則將其替𠀀𠀀 -2。
- 特定值 sys.hash_info.inf 和 -sys.hash_info.inf （分𠀀）被用作正無窮大或負無窮大的雜𠀀值。
- 對於一個*complex* 值 z，實部和𠀀部的雜𠀀值藉由 hash(z.real) + sys.hash_info.imag * hash(z.imag) 的計算進行組合，對 2**sys.hash_info.width 取模數使其介於 range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))。同樣地，如果結果是 -1，則將其替𠀀𠀀 -2。

𠀀了闡明上述規則，這𠀀有一些 Python 程式碼範例，等同於𠀀建的雜𠀀，用於計算有理數、*float* 或*complex* 的雜𠀀：

```
import sys, math

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

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

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

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

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

    if math.isnan(x):
```

(繼續下一頁)

(繼續上一頁)

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

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

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

4.5 Boolean 型🄵 - `bool`

Boolean 值代表 truth values（真值）。*bool* 型🄵有兩個常數實例：`True` 和 `False`。

🄵建函式*bool()* 將任何值轉🄵🄵 boolean 值，如果該值可以被直譯🄵一個 truth value（真值）（見上面的真值檢測章節）。

對於邏輯運算，使用*boolean 運算子* `and`、`or` 和 `not`。當將位元運算子 `&`、`|`、`^` 應用於兩個 boolean 值時，它們會回傳一個等同於邏輯運算"and"、"or"、"xor" 的 boolean 值。然而，應該優先使用邏輯運算子 `and`、`or` 和 `!=` 而不是 `&`、`|` 和 `^`。

在 3.12 版之後被🄵用: 位元反轉運算子 ~ 的使用已被🄵用🄵且將在 Python 3.16 中引發錯誤。

bool 是*int* 的子類🄵（見*數值型🄵 --- int、float、complex*）。在許多數值情境中，`False` 和 `True` 分🄵像整數 0 和 1 一樣。然而，不鼓勵依賴這一點；請使用*int()* 進行顯式轉🄵。

4.6 🄵代器型🄵

Python 支援對容器的🄵代概念。這是實作兩種不同的 methods；這些方法被用於允許使用者自定義的 classes 以支援🄵代。序列則總是支援這些🄵代 methods，在下方有更詳細的描述。

需要🄵容器物件定義一個 method 來提供*可🄵代物件*支援：

`container.__iter__()`

> 回傳一個*🄵代器*物件。該物件需要支援下述的🄵代器協定。如果一個容器支援不同型🄵的🄵代，則可以提供額外的 methods 來專門請求這些🄵代型🄵的🄵代器。（支援多種形式🄵代的物件的一個例子是支援廣度優先和深度優先遍歷的樹結構。）此 method 對應 Python/C API 中 Python 物件的型🄵結構的 `tp_iter` 插槽。

🄵代器物件本身需要支援下列兩個 methods，他們一起形成了 *🄵代器協定*：

`iterator.__iter__()`

> 回傳*🄵代器*物件本身。這是🄵了允許容器和🄵代器都可以與 `for` 和 `in` 在陳述式中使用。此 method 對應於 Python/C API 中 Python 物件的型🄵結構的 `tp_iter` 插槽。

`iterator.__next__()`

> 從*🄵代器*回傳下一個項目。如果🄵有更多項目，則引發*StopIteration* 例外。此 method 對應於 Python/C API 中 Python 物件的型🄵結構的 `tp_iternext` 插槽。

Python 定義了幾個🄵代器物件來支援對一般和特定序列型🄵、字典和其他更專門的形式的🄵代。這些特定型🄵除了實作🄵代器協定外🄵不重要。

一旦🄵代器的*__next__()* method 引發*StopIteration*，則它必須在後續呼叫中繼續這樣做。不遵守此屬性的實作被認🄵是有問題的。

4.6.1 Generator Types

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

4.7 Sequence Types --- `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of *binary data* and *text strings* are described in dedicated sections.

4.7.1 Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.[3]

運算	結果	🄵解
`x in s`	如果 *s* 的一個項目等於 *x* 則🄵 `True`，否則🄵 `False`	(1)
`x not in s`	如果 *s* 的一個項目等於 *x* 則🄵 `False`，否則🄵 `True`	(1)
`s + t`	*s* 和 *t* 的串接	(6)(7)
`s * n` 或 `n * s`	等同於將 *s* 加到自己 *n* 次	(2)(7)
`s[i]`	*s* 的第 *i* 項，起始🄵 0	(3)(8)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	*s* 的長度	
`min(s)`	*s* 中最小的項目	
`max(s)`	*s* 中最大的項目	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Forward and reversed iterators over mutable sequences access values using an index. That index will continue to march forward (or backward) even if the underlying sequence is mutated. The iterator terminates only when an `IndexError` or a `StopIteration` is encountered (or when the index drops below zero).

🄵解:

(1) While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as `str`, `bytes` and `bytearray`) also use them for subsequence testing:

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

(2) Values of *n* less than `0` are treated as `0` (which yields an empty sequence of the same type as *s*). Note that items in the sequence *s* are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

[3] They must have since the parser can't tell the type of the operands.

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

What has happened is that `[[]]` is a one-element list containing an empty list, so all three elements of `[[]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

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

Further explanation is available in the FAQ entry faq-multidimensional-list.

(3) If *i* or *j* is negative, the index is relative to the end of sequence *s*: `len(s) + i` or `len(s) + j` is substituted. But note that `-0` is still `0`.

(4) The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that `i <= k < j`. If *i* or *j* is greater than `len(s)`, use `len(s)`. If *i* is omitted or `None`, use `0`. If *j* is omitted or `None`, use `len(s)`. If *i* is greater than or equal to *j*, the slice is empty.

(5) The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index `x = i + n*k` such that `0 <= n < (j-i)/k`. In other words, the indices are `i`, `i+k`, `i+2*k`, `i+3*k` and so on, stopping when *j* is reached (but never including *j*). When *k* is positive, *i* and *j* are reduced to `len(s)` if they are greater. When *k* is negative, *i* and *j* are reduced to `len(s) - 1` if they are greater. If *i* or *j* are omitted or `None`, they become ”end” values (which end depends on the sign of *k*). Note, *k* cannot be zero. If *k* is `None`, it is treated like `1`.

(6) Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:

- if concatenating *`str`* objects, you can build a list and use *`str.join()`* at the end or else write to an *`io.StringIO`* instance and retrieve its value when complete
- if concatenating *`bytes`* objects, you can similarly use *`bytes.join()`* or *`io.BytesIO`*, or you can do in-place concatenation with a *`bytearray`* object. *`bytearray`* objects are mutable and have an efficient overallocation mechanism
- if concatenating *`tuple`* objects, extend a *`list`* instead
- for other types, investigate the relevant class documentation

(7) Some sequence types (such as *`range`*) only support item sequences that follow specific patterns, and hence don't support sequence concatenation or repetition.

(8) An *`IndexError`* is raised if *i* is outside the sequence range.

序列方法

序列型🄵也支援以下方法：

`sequence.`**`count`**`(`*value*`, /)`

回傳 *sequence* 中 *value* 出現的總次數。

`sequence.`**`index`**`(`*value[, start[, stop]*`)`

回傳 *sequence* 中 *value* 首次出現的索引。

當 *value* 不在 *sequence* 中時引發*`ValueError`*。

The *start* or *stop* arguments allow for efficient searching of subsections of the sequence, beginning at *start* and ending at *stop*. This is roughly equivalent to `start + sequence[start:stop].index(value)`, only without copying any data.

> **警示**
>
> Not all sequence types support passing the *start* and *stop* arguments.

4.7.2 不可變序列型🄵

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the *hash()* built-in.

This support allows immutable sequences, such as *tuple* instances, to be used as *dict* keys and stored in *set* and *frozenset* instances.

Attempting to hash an immutable sequence that contains unhashable values will result in *TypeError*.

4.7.3 可變序列型🄵

The operations in the following table are defined on mutable sequence types. The *collections.abc.MutableSequence* ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table *s* is an instance of a mutable sequence type, *t* is any iterable object and *x* is an arbitrary object that meets any type and value restrictions imposed by *s* (for example, *bytearray* only accepts integers that meet the value restriction `0 <= x <= 255`).

運算	結果	🄵解
`s[i] = x`	item *i* of *s* is replaced by *x*	
`del s[i]`	移除 *s* 中的項目 *i*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	移除串列中 `s[i:j]` 的元素（和 `s[i:j] = []` 相同）	
`s[i:j:k] = t`	`s[i:j:k]` 的元素被 *t* 的元素取代	(1)
`del s[i:j:k]`	移除串列中 `s[i:j:k]` 的元素	
`s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)	
`s *= n`	updates *s* with its contents repeated *n* times	(2)

🄵解:

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

(2) The value *n* is an integer, or an object implementing `__index__()`. Zero and negative values of *n* clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under *Common Sequence Operations*.

可變序列方法

可變序列型🄵也支援以下方法:

`sequence.`**`append`**`(`*value*`, /)`

Append *value* to the end of the sequence This is equivalent to writing `seq[len(seq):len(seq)] = [value]`.

sequence.**clear**()

在 3.3 版被加入.

移除 *sequence* 中的所有項目，這和 `del sequence[:]` 相同。

sequence.**copy**()

在 3.3 版被加入.

建立 *sequence* 的淺層🄵🄵，這和 `sequence[:]` 相同。

> **提示**
>
> The `copy()` method is not part of the *MutableSequence ABC*, but most concrete mutable sequence types provide it.

sequence.**extend**(*iterable*, */*)

Extend *sequence* with the contents of *iterable*. For the most part, this is the same as writing `seq[len(seq):len(seq)] = iterable`.

sequence.**insert**(*index*, *value*, */*)

Insert *value* into *sequence* at the given *index*. This is equivalent to writing `sequence[index:index] = [value]`.

sequence.**pop**(*index=-1*, */*)

Retrieve the item at *index* and also removes it from *sequence*. By default, the last item in *sequence* is removed and returned.

sequence.**remove**(*value*, */*)

Remove the first item from *sequence* where `sequence[i] == value`.

當 *value* 不在 *sequence* 中時引發*ValueError*。

sequence.**reverse**()

Reverse the items of *sequence* in place. This method maintains economy of space when reversing a large sequence. To remind users that it operates by side-effect, it returns `None`.

4.7.4 List（串列）

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

class list(*iterable=()*, */*)

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
- 使用串列綜合運算式：`[x for x in iterable]`
- 使用型🄵建構函式：`list()` 或 `list(iterable)`

The constructor builds a list whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[]`.

Many other operations also produce lists, including the *sorted()* built-in.

Lists implement all of the *common* and *mutable* sequence operations. Lists also provide the following additional method:

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

This method sorts the list in place, using only < comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

sort() accepts two arguments that can only be passed by keyword (*keyword-only arguments*):

key specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The *functools.cmp_to_key()* utility is available to convert a 2.x style *cmp* function to a *key* function.

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use *sorted()* to explicitly request a new sorted list instance).

The *sort()* method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal --- this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see sortinghowto.

CPython 實作細節： While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises *ValueError* if it can detect that the list has been mutated during a sort.

4.7.5 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the *enumerate()* built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a *set* or *dict* instance).

class tuple(*iterable=()*, */*)

元組可以以多種方式建構：

- 使用一對圓括號表示空元組：`()`
- 使用末尾的逗號表示單元素元組：`a,` 或 `(a,)`
- 使用逗號分隔項目：`a, b, c` 或 `(a, b, c)`
- Using the *tuple()* built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple( [1, 2, 3] )` returns `(1, 2, 3)`. If no argument is given, the constructor creates a new empty tuple, `()`.

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, `f(a, b, c)` is a function call with three arguments, while `f((a, b, c))` is a function call with a 3-tuple as the sole argument.

Tuples implement all of the *common* sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, *collections.namedtuple()* may be a more appropriate choice than a simple tuple object.

4.7.6 Ranges

The *range* type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

class range(*stop*, */*)

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

The arguments to the range constructor must be integers (either built-in *int* or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, *ValueError* is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than *sys.maxsize* are permitted but some features (such as *len()*) may raise *OverflowError*.

Range examples:

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

Ranges implement all of the *common* sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

start

The value of the *start* parameter (or `0` if the parameter was not supplied)

stop

The value of the *stop* parameter

step

The value of the *step* parameter (or `1` if the parameter was not supplied)

The advantage of the *range* type over a regular *list* or *tuple* is that a *range* object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Range objects implement the *collections.abc.Sequence* ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see *Sequence Types --- list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
```

(繼續下一頁)

(繼續上一頁)

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

Testing range objects for equality with `==` and `!=` compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different *start*, *stop* and *step* attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

在 3.2 版的變更: Implement the Sequence ABC. Support slicing and negative indices. Test *int* objects for membership in constant time instead of iterating through all items.

在 3.3 版的變更: Define '==' and '!=' to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

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

也參考

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

4.8 Text and Binary Sequence Type Methods Summary

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

分類		*str* 方法	
格式化		*str.format()*	
		str.format_map()	
		f-strings	
		printf-style String Formatting	
搜尋和取代	*str.find()*	*str.rfind()*	
	str.index()	*str.rindex()*	
		str.startswith()	
		str.endswith()	
		str.count()	
		str.replace()	
分割和連接	*str.split()*		*str.rsplit()*
		str.splitlines()	
		str.partition()	
		str.rpartition()	
		str.join()	
字串分類		*str.isalpha()*	
		str.isdecimal()	
		str.isdigit()	
		str.isnumeric()	
		str.isalnum()	
		str.isidentifier()	

分類					str 方法
					str.islower()
					str.isupper()
					str.istitle()
					str.isspace()
					str.isprintable()
大小寫操作					str.lower()
					str.upper()
					str.casefold()
					str.capitalize()
					str.title()
					str.swapcase()
填充和去除空白	str.ljust()				str.rjust()
					str.center()
					str.expandtabs()
					str.strip()
		str.lstrip()			
翻譯和編碼					str.translate()
					str.maketrans()
					str.encode()

4.9 Text Sequence Type --- str

Textual data in Python is handled with str objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, `("spam " "eggs") == "spam eggs"`.

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

Strings may also be created from other objects using the str constructor.

Since there is no separate "character" type, indexing a string produces strings of length 1. That is, for a non-empty string *s*, `s[0] == s[0:1]`.

There is also no mutable string type, but str.join() or io.StringIO can be used to efficiently construct strings from multiple fragments.

在 3.3 版的變更: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

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

class str(*object*)

class str(*object*, *encoding*, *errors='strict'*)

class str(*object*, *, *errors*)

Return a *string* version of *object*. If *object* is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether *encoding* or *errors* is given, as follows.

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

If at least one of *encoding* or *errors* is given, *object* should be a *bytes-like object* (e.g. *bytes* or *bytearray*). In this case, if *object* is a *bytes* (or *bytearray*) object, then `str(bytes, encoding, errors)` is equivalent to *bytes.decode(encoding, errors)*. Otherwise, the bytes object underlying the buffer object is obtained before calling *bytes.decode()*. See *Binary Sequence Types --- bytes, bytearray, memoryview* and bufferobjects for information on buffer objects.

Passing a *bytes* object to *str()* without the *encoding* or *errors* arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

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

For more information on the `str` class and its methods, see *Text Sequence Type --- str* and the 字串方法 section below. To output formatted strings, see the f-strings and 格式化文字語法 sections. In addition, see the 文本處理 *(Text Processing)* 服務 section.

4.9.1 字串方法

Strings implement all of the *common* sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see *str.format()*, 格式化文字語法 and 自訂字串格式) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (*printf-style String Formatting*).

The 文本處理 *(Text Processing)* 服務 section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the *re* module).

`str.`**`capitalize`**`()`

Return a copy of the string with its first character capitalized and the rest lowercased.

在 3.8 版的變更: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.`**`casefold`**`()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, *lower()* would do nothing to `'ß'`; *casefold()* converts it to `"ss"`.

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

在 3.3 版被加入.

`str.`**`center`**`(`*width*, *fillchar*=' ', /`)`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`. For example:

```
>>> 'Python'.center(10)
'  Python  '
>>> 'Python'.center(10, '-')
'--Python--'
>>> 'Python'.center(4)
'Python'
```

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

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

If *sub* is empty, returns the number of empty strings between characters which is the length of the string plus one. For example:

```
>>> 'spam, spam, spam'.count('spam')
3
>>> 'spam, spam, spam'.count('spam', 5)
2
>>> 'spam, spam, spam'.count('spam', 5, 10)
1
>>> 'spam, spam, spam'.count('eggs')
0
>>> 'spam, spam, spam'.count('')
17
```

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

Return the string encoded to *bytes*.

encoding defaults to `'utf-8'`; see 標準編碼 for possible values.

errors controls how encoding errors are handled. If `'strict'` (the default), a *UnicodeError* exception is raised. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via *codecs.register_error()*. See *Error Handlers* for details.

For performance reasons, the value of *errors* is not checked for validity unless an encoding error actually occurs, *Python 開發模式* is enabled or a debug build is used. For example:

```
>>> encoded_str_to_bytes = 'Python'.encode()
>>> type(encoded_str_to_bytes)
<class 'bytes'>
>>> encoded_str_to_bytes
b'Python'
```

在 3.1 版的變更: 新增關鍵字引數的支援。

在 3.9 版的變更: The value of the *errors* argument is now checked in *Python 開發模式* and in debug mode.

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

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position. Using *start* and *end* is equivalent to `str[start:end].endswith(suffix)`. For example:

```
>>> 'Python'.endswith('on')
True
>>> 'a tuple of suffixes'.endswith(('at', 'in'))
False
>>> 'a tuple of suffixes'.endswith(('at', 'es'))
True
>>> 'Python is amazing'.endswith('is', 0, 9)
True
```

另請參🄵*startswith()* 和*removesuffix()*。

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

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed. For example:

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
>>> print('01\t012\n0123\t01234'.expandtabs(4))
01  012
0123    01234
```

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

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found. For example:

```
>>> 'spam, spam, spam'.find('sp')
0
>>> 'spam, spam, spam'.find('sp', 5)
6
```

另請參F*rfind()* 和*index()*。

備F

The *find()* method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

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

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

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

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

See 格式化文字語法 for a description of the various formatting options that can be specified in format strings.

備F

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

在 3.7 版的變更: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

str.**format_map**(*mapping*, */*)

Similar to `str.format(**mapping)`, except that `mapping` is used directly and not copied to a *dict*. This is useful if for example `mapping` is a dict subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

在 3.2 版被加入.

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

Like *find()*, but raise *ValueError* when the substring is not found.

str.**isalnum**()

Return `True` if all characters in the string are alphanumeric and there is at least one character, `False` otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

str.**isalpha**()

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

str.**isascii**()

Return `True` if the string is empty or all characters in the string are ASCII, `False` otherwise. ASCII characters have code points in the range U+0000-U+007F.

在 3.7 版被加入.

str.**isdecimal**()

Return `True` if all characters in the string are decimal characters and there is at least one character, `False` otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category "Nd".

str.**isdigit**()

Return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.**isidentifier**()

Return `True` if the string is a valid identifier according to the language definition, section identifiers.

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

範例:

```
>>> from keyword import iskeyword

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

str.**islower**()

Return `True` if all cased characters[4] in the string are lowercase and there is at least one cased character, `False` otherwise.

str.**isnumeric**()

Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

[4] Cased characters are those with general category property being one of "Lu" (Letter, uppercase), "Ll" (Letter, lowercase), or "Lt" (Letter, titlecase).

str.**isprintable**()

Return `True` if all characters in the string are printable, `False` if it contains at least one non-printable character.

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

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

str.**isspace**()

Return `True` if there are only whitespace characters in the string and there is at least one character, `False` otherwise.

A character is *whitespace* if in the Unicode character database (see *unicodedata*), either its general category is `Zs` (”Separator, space”), or its bidirectional class is one of `WS`, `B`, or `S`.

str.**istitle**()

Return `True` if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return `False` otherwise.

str.**isupper**()

Return `True` if all cased characters[Page 57, 4] in the string are uppercase and there is at least one cased character, `False` otherwise.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.**join**(*iterable*, */*)

Return a string which is the concatenation of the strings in *iterable*. A *TypeError* will be raised if there are any non-string values in *iterable*, including *bytes* objects. The separator between elements is the string providing this method.

str.**ljust**(*width*, *fillchar=' '*, */*)

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.**lower**()

Return a copy of the string with all the cased characters[Page 57, 4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 ’Default Case Folding’ of the Unicode Standard.

str.**lstrip**(*chars=None*, */*)

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

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

See *str.removeprefix()* for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static str.**maketrans**(*dict*, /)

static str.**maketrans**(*from*, *to*, *remove*='', /)

This static method returns a translation table usable for *str.translate()*.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *from* will be mapped to the character at the same position in *to*. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

str.**partition**(*sep*, /)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.**removeprefix**(*prefix*, /)

If the string starts with the *prefix* string, return `string[len(prefix):]`. Otherwise, return a copy of the original string:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

在 3.9 版被加入.

str.**removesuffix**(*suffix*, /)

If the string ends with the *suffix* string and that *suffix* is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

在 3.9 版被加入.

str.**replace**(*old*, *new*, /, *count*=*-1*)

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced. If *count* is not specified or `-1`, then all occurrences are replaced.

在 3.13 版的變更: *count* 現在作🄴關鍵字引數被支援。

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

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

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

Like *rfind()* but raises *ValueError* when the substring *sub* is not found.

str.**rjust**(*width*, *fillchar*=' ', /)

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.**rpartition**(*sep*, /)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

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

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, *rsplit()* behaves like *split()* which is described in detail below.

str.**rstrip**(*chars=None*, /)

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

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

See *str.removesuffix()* for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

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

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use *re.split()*). Splitting an empty string with a specified separator returns `['']`.

舉例來F:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

舉例來F:

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

If *sep* is not specified or is `None` and *maxsplit* is `0`, only leading runs of consecutive whitespace are considered.

舉例來🄵:

```
>>> "".split(None, 0)
[]
>>> "   ".split(None, 0)
[]
>>> "   foo   ".split(maxsplit=0)
['foo   ']
```

`str.`**`splitlines`**(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of *universal newlines*.

Representation	描述
`\n`	Line Feed
`\r`	Carriage Return
`\r\n`	Carriage Return + Line Feed
`\v` 或 `\x0b`	Line Tabulation
`\f` 或 `\x0c`	Form Feed
`\x1c`	File Separator
`\x1d`	Group Separator
`\x1e`	Record Separator
`\x85`	Next Line (C1 Control Code)
`\u2028`	Line Separator
`\u2029`	Paragraph Separator

在 3.2 版的變更: `\v` and `\f` added to list of line boundaries.

舉例來🄵:

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

Unlike *`split()`* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

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

For comparison, `split('\n')` gives:

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

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

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

str.**strip**(*chars=None*, */*)

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

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

The outermost leading and trailing *chars* argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in *chars*. A similar action takes place on the trailing end. For example:

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

str.**swapcase**()

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

str.**title**()

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

舉例來🄵:

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

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

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

The *string.capwords()* function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

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

str.**translate**(*table*, */*)

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a *mapping* or *sequence*. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a *LookupError* exception, to map the character to itself.

You can use *str.maketrans()* to create a translation map from character-to-character mappings in different formats.

See also the *codecs* module for a more flexible approach to custom character mappings.

`str.upper()`

Return a copy of the string with all the cased characters[Page 57, 4] converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not "Lu" (Letter, uppercase), but e.g. "Lt" (Letter, titlecase).

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

`str.zfill(`*width*`, /)`

Return a copy of the string left filled with ASCII `'0'` digits to make a string of length *width*. A leading sign prefix (`'+'`/`'-'`) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

舉例來🄵:

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

4.9.2 Formatted String Literals (f-strings)

在 3.6 版被加入.

在 3.7 版的變更: The `await` and `async for` can be used in expressions within f-strings.

在 3.8 版的變更: Added the debugging operator (=)

在 3.12 版的變更: Many restrictions on expressions within f-strings have been removed. Notably, nested strings, comments, and backslashes are now permitted.

An *f-string* (formally a *formatted string literal*) is a string literal that is prefixed with `f` or `F`. This type of string literal allows embedding arbitrary Python expressions within *replacement fields*, which are delimited by curly brackets (`{}`). These expressions are evaluated at runtime, similarly to *`str.format()`*, and are converted into regular *`str`* objects. For example:

```
>>> who = 'nobody'
>>> nationality = 'Spanish'
>>> f'{who.title()} expects the {nationality} Inquisition!'
'Nobody expects the Spanish Inquisition!'
```

It is also possible to use a multi line f-string:

```
>>> f'''This is a string
... on two lines'''
'This is a string\non two lines'
```

A single opening curly bracket, `'{'`, marks a *replacement field* that can contain any Python expression:

```
>>> nationality = 'Spanish'
>>> f'The {nationality} Inquisition!'
'The Spanish Inquisition!'
```

To include a literal `{` or `}`, use a double bracket:

```
>>> x = 42
>>> f'{{x}} is {x}'
'{x} is 42'
```

Functions can also be used, and *format specifiers*:

```
>>> from math import sqrt
>>> f'√2 \N{ALMOST EQUAL TO} {sqrt(2):.5f}'
'√2 ≈ 1.41421'
```

Any non-string expression is converted using *str()*, by default:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)}'
'1/3'
```

To use an explicit conversion, use the `!` (exclamation mark) operator, followed by any of the valid formats, which are:

Conversion	含義
`!a`	*ascii()*
`!r`	*repr()*
`!s`	*str()*

舉例來🄵:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)!s}'
'1/3'
>>> f'{Fraction(1, 3)!r}'
'Fraction(1, 3)'
>>> question = '¿Dónde está el Presidente?'
>>> print(f'{question!a}')
'\xbfD\xf3nde est\xe1 el Presidente?'
```

While debugging it may be helpful to see both the expression and its value, by using the equals sign (=) after the expression. This preserves spaces within the brackets, and can be used with a converter. By default, the debugging operator uses the *repr()* (`!r`) conversion. For example:

```
>>> from fractions import Fraction
>>> calculation = Fraction(1, 3)
>>> f'{calculation=}'
'calculation=Fraction(1, 3)'
>>> f'{calculation = }'
'calculation = Fraction(1, 3)'
>>> f'{calculation = !s}'
'calculation = 1/3'
```

Once the output has been evaluated, it can be formatted using a *format specifier* following a colon (`':'`). After the expression has been evaluated, and possibly converted to a string, the `__format__()` method of the result is called with the format specifier, or the empty string if no format specifier is given. The formatted result is then used as the final value for the replacement field. For example:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 7):.6f}'
'0.142857'
>>> f'{Fraction(1, 7):_^+10}'
'___+1/7___'
```

4.9.3 `printf`-style String Formatting

備🄵

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly).

Using formatted string literals, the *str.format()* interface, or *string.Template* may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` function in the C language. For example:

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

If *format* requires a single argument, *values* may be a single non-tuple object.[5] Otherwise, *values* must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

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

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	含義
`'#'`	The value conversion will use the ”alternate form” (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a ”space” flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python -- so e.g. `%ld` is identical to `%d`.

The conversion types are:

[5] To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

Con-version	含義	𝔽解
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type -- it is identical to `'d'`.	(6)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single character (accepts integer or single character string).	
`'r'`	String (converts any Python object using *repr()*).	(5)
`'s'`	String (converts any Python object using *str()*).	(5)
`'a'`	String (converts any Python object using *ascii()*).	(5)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

𝔽解:

(1) The alternate form causes a leading octal specifier (`'0o'`) to be inserted before the first digit.

(2) The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted before the first digit.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) If precision is `N`, the output is truncated to `N` characters.

(6) 參𝔽 **PEP 237**。

Since Python strings have an explicit length, `%s` conversions do not assume that `'\0'` is the end of the string.

在 3.1 版的變更: `%f` conversions for numbers whose absolute value is over 1e50 are no longer replaced by `%g` conversions.

4.10 Binary Sequence Types --- `bytes`, `bytearray`, `memoryview`

The core built-in types for manipulating binary data are *bytes* and *bytearray*. They are supported by *memoryview* which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The *array* module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

4.10.1 Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

class bytes(*source=b''*)

class bytes(*source*, *encoding*, *errors='strict'*)

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a `b` prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- Triple quoted: `b'''3 single quotes'''`, `b"""3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See strings for more about the various forms of bytes literal, including supported escape sequences.

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

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

另見🄵建的*bytes*。

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod fromhex(*string*, */*)

This *`bytes`* class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

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

在 3.7 版的變更: *`bytes.fromhex()`* now skips all ASCII whitespace in the string, not just spaces.

在 3.14 版的變更: *`bytes.fromhex()`* now accepts ASCII *`bytes`* and *bytes-like objects* as input.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

hex(***, *bytes_per_sep=1*)

hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the instance.

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

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default, this separator will be included between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

在 3.5 版被加入.

在 3.8 版的變更: *bytes.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

4.10.2 Bytearray 物件

bytearray objects are a mutable counterpart to *bytes* objects.

class bytearray(*source=b''*)

class bytearray(*source*, *encoding*, *errors='strict'*)

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- 建立一個空的實例: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi!')`

As bytearray objects are mutable, they support the *mutable* sequence operations in addition to the common bytes and bytearray operations described in *Bytes 和 Bytearray 的操作*.

另見🄵建的*bytearray*。

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod fromhex(*string*, */*)

This *bytearray* class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

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

在 3.7 版的變更: *bytearray.fromhex()* now skips all ASCII whitespace in the string, not just spaces.

在 3.14 版的變更: *bytearray.fromhex()* now accepts ASCII *bytes* and *bytes-like objects* as input.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

hex(*, *bytes_per_sep=1*)

hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the instance.

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

在 3.5 版被加入.

在 3.8 版的變更: Similar to *bytes.hex()*, *bytearray.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

resize(*size*, */*)

Resize the *bytearray* to contain *size* bytes. *size* must be greater than or equal to 0.

If the *bytearray* needs to shrink, bytes beyond *size* are truncated.

If the *bytearray* needs to grow, all new bytes, those beyond *size*, will be set to null bytes.

等同於:

```
>>> def resize(ba, size):
...     if len(ba) > size:
...         del ba[size:]
...     else:
...         ba += b'\0' * (size - len(ba))
```

範例:

```
>>> shrink = bytearray(b'abc')
>>> shrink.resize(1)
>>> (shrink, len(shrink))
(bytearray(b'a'), 1)
>>> grow = bytearray(b'abc')
>>> grow.resize(5)
>>> (grow, len(grow))
(bytearray(b'abc\x00\x00'), 5)
```

在 3.14 版被加入.

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert a bytearray object into a list of integers using `list(b)`.

4.10.3 Bytes 和 Bytearray 的操作

Both bytes and bytearray objects support the *common* sequence operations. They interoperate not just with operands of the same type, but with any *bytes-like object*. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

備🄵

The methods on bytes and bytearray objects don't accept strings as their arguments, just as the methods on strings don't accept bytes as their arguments. For example, you have to write:

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

和:

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

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

> **備F**
>
> Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

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

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

Return the number of non-overlapping occurrences of subsequence *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

If *sub* is empty, returns the number of empty slices between characters which is the length of the bytes object plus one.

在 3.3 版的變更: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.`**`removeprefix`**(*prefix*, /)

`bytearray.`**`removeprefix`**(*prefix*, /)

If the binary data starts with the *prefix* string, return `bytes[len(prefix):]`. Otherwise, return a copy of the original binary data:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

prefix 可以是任何的*bytes-like object*。

> **備F**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

在 3.9 版被加入.

`bytes.`**`removesuffix`**(*suffix*, /)

`bytearray.`**`removesuffix`**(*suffix*, /)

If the binary data ends with the *suffix* string and that *suffix* is not empty, return `bytes[:-len(suffix)]`. Otherwise, return a copy of the original binary data:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

The *suffix* may be any *bytes-like object*.

> **備F**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

在 3.9 版被加入.

bytes.**decode**(*encoding='utf-8'*, *errors='strict'*)

bytearray.**decode**(*encoding='utf-8'*, *errors='strict'*)

Return the bytes decoded to a *str*.

encoding defaults to `'utf-8'`; see 標準編碼 for possible values.

errors controls how decoding errors are handled. If `'strict'` (the default), a *UnicodeError* exception is raised. Other possible values are `'ignore'`, `'replace'`, and any other name registered via *codecs.register_error()*. See *Error Handlers* for details.

For performance reasons, the value of *errors* is not checked for validity unless a decoding error actually occurs, *Python 開發模式* is enabled or a debug build is used.

> **備🄵**
>
> Passing the *encoding* argument to *str* allows decoding any *bytes-like object* directly, without needing to make a temporary `bytes` or `bytearray` object.

在 3.1 版的變更: 新增關鍵字引數的支援。

在 3.9 版的變更: The value of the *errors* argument is now checked in *Python 開發模式* and in debug mode.

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

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

Return `True` if the binary data ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The suffix(es) to search for may be any *bytes-like object*.

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

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

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

> **備🄵**
>
> The *find()* method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

在 3.3 版的變更: Also accept an integer in the range 0 to 255 as the subsequence.

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

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

Like *find()*, but raise *ValueError* when the subsequence is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

在 3.3 版的變更: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.**join**(*iterable*, /)

bytearray.**join**(*iterable*, /)

Return a bytes or bytearray object which is the concatenation of the binary data sequences in *iterable*. A `TypeError` will be raised if there are any values in *iterable* that are not *bytes-like objects*, including `str` objects. The separator between elements is the contents of the bytes or bytearray object providing this method.

static bytes.**maketrans**(*from*, *to*, /)

static bytearray.**maketrans**(*from*, *to*, /)

This static method returns a translation table usable for `bytes.translate()` that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be *bytes-like objects* and have the same length.

在 3.1 版被加入.

bytes.**partition**(*sep*, /)

bytearray.**partition**(*sep*, /)

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

The separator to search for may be any *bytes-like object*.

bytes.**replace**(*old*, *new*, *count=-1*, /)

bytearray.**replace**(*old*, *new*, *count=-1*, /)

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any *bytes-like object*.

> **備F**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

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

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

Return the highest index in the sequence where the subsequence *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

在 3.3 版的變更: Also accept an integer in the range 0 to 255 as the subsequence.

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

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

Like `rfind()` but raises `ValueError` when the subsequence *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

在 3.3 版的變更: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.**rpartition**(*sep*, /)

bytearray.**rpartition**(*sep*, /)

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty bytes or bytearray objects, followed by a copy of the original sequence.

The separator to search for may be any *bytes-like object*.

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

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

Return `True` if the binary data starts with the specified *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The prefix(es) to search for may be any *bytes-like object*.

bytes.**translate**(*table*, */*, *delete=b''*)

bytearray.**translate**(*table*, */*, *delete=b''*)

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the *table* argument to `None` for translations that only delete characters:

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

在 3.6 版的變更: 支援 *delete* 關鍵字引數。

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.**center**(*width*, *fillbyte=b' '*, */*)

bytearray.**center**(*width*, *fillbyte=b' '*, */*)

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**ljust**(*width*, *fillbyte=b' '*, */*)

bytearray.**ljust**(*width*, *fillbyte=b' '*, */*)

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**lstrip**(*bytes=None*, */*)

bytearray.**lstrip**(*bytes=None*, */*)

Return a copy of the sequence with specified leading bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix; rather, all combinations of its values are stripped:

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

The binary sequence of byte values to remove may be any *bytes-like object*. See `removeprefix()` for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

> **備𝔽**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`rjust`**(*width*, *fillbyte=b' '*, */*)

`bytearray.`**`rjust`**(*width*, *fillbyte=b' '*, */*)

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

> **備𝔽**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

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

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

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`bytes.`**`rstrip`**(*bytes=None*, */*)

`bytearray.`**`rstrip`**(*bytes=None*, */*)

Return a copy of the sequence with specified trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a suffix; rather, all combinations of its values are stripped:

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

The binary sequence of byte values to remove may be any *bytes-like object*. See `removesuffix()` for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

> **備𝔽**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

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

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

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or is `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence as a single delimiter. Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

舉例來🄵:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

舉例來🄵:

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

bytes.**strip**(*bytes=None*, */*)

bytearray.**strip**(*bytes=None*, */*)

Return a copy of the sequence with specified leading and trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

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

The binary sequence of byte values to remove may be any *bytes-like object*.

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytearray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.**capitalize**()

bytearray.**capitalize**()

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**expandtabs**(*tabsize=8*)

bytearray.**expandtabs**(*tabsize=8*)

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (`b'\t'`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (`b'\n'`) or carriage return (`b'\r'`), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

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

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**isalnum**()

bytearray.**isalnum**()

Return `True` if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

舉例來🄵:

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

bytes.**isalpha**()

bytearray.**isalpha**()

Return `True` if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

舉例來🄵:

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

(繼續下一頁)

(繼續上一頁)

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

bytes.**isascii**()

bytearray.**isascii**()

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

在 3.7 版被加入.

bytes.**isdigit**()

bytearray.**isdigit**()

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

舉例來🄵:

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

bytes.**islower**()

bytearray.**islower**()

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

舉例來🄵:

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

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**isspace**()

bytearray.**isspace**()

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

bytes.**istitle**()

bytearray.**istitle**()

Return `True` if the sequence is ASCII titlecase and the sequence is not empty, `False` otherwise. See *bytes.title()* for more details on the definition of "titlecase".

舉例來🄵:

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

bytes.**isupper**()

bytearray.**isupper**()

Return `True` if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, `False` otherwise.

舉例來🄵:

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

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**lower**()

bytearray.**lower**()

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

舉例來𝔽:

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

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

備𝔽

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**splitlines**(*keepends=False*)

bytearray.**splitlines**(*keepends=False*)

Return a list of the lines in the binary sequence, breaking at ASCII line boundaries. This method uses the *universal newlines* approach to splitting lines. Line breaks are not included in the resulting list unless *keepends* is given and true.

舉例來𝔽:

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

Unlike *split()* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

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

bytes.**swapcase**()

bytearray.**swapcase**()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

舉例來𝔽:

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

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike *str.swapcase()*, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**title**()

bytearray.**title**()

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

舉例來🄵:

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

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

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

A workaround for apostrophes can be constructed using regular expressions:

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

> **備🄵**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.**upper**()

bytearray.**upper**()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

舉例來🄵:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

> **備F**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.`**`zfill`**`(`*width*, /`)`

`bytearray.`**`zfill`**`(`*width*, /`)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length *width*. A leading sign prefix (`b'+'`/ `b'-'`) is handled by inserting the padding *after* the sign character rather than before. For *bytes* objects, the original sequence is returned if *width* is less than or equal to `len(seq)`.

舉例來F:

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

> **備F**
>
> The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

4.10.4 `printf`-style Bytes Formatting

> **備F**
>
> The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (`bytes`/`bytearray`) have one unique built-in operation: the `%` operator (modulo). This is also known as the bytes *formatting* or *interpolation* operator. Given `format % values` (where *format* is a bytes object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` in the C language.

If *format* requires a single argument, *values* may be a single non-tuple object.[Page 65, 5] Otherwise, *values* must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).

7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

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

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	含義
`'#'`	The value conversion will use the ”alternate form” (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a ”space” flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python -- so e.g. `%ld` is identical to `%d`.

The conversion types are:

Conversion	含義	🄵解
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type -- it is identical to `'d'`.	(8)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single byte (accepts integer or single byte objects).	
`'b'`	Bytes (any object that follows the buffer protocol or has `__bytes__()`).	(5)
`'s'`	`'s'` is an alias for `'b'` and should only be used for Python2/3 code bases.	(6)
`'a'`	Bytes (converts any Python object using `repr(obj).encode('ascii', 'backslashreplace')`).	(5)
`'r'`	`'r'` is an alias for `'a'` and should only be used for Python2/3 code bases.	(7)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

🄵解:

(1) The alternate form causes a leading octal specifier (`'0o'`) to be inserted before the first digit.

(2) The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted before the first digit.

(3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.

The precision determines the number of digits after the decimal point and defaults to 6.

(4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

(5) If precision is `N`, the output is truncated to `N` characters.

(6) `b'%s'` 已被🄵用，但在 3.x 系列中不會被移除。

(7) `b'%r'` 已被🄵用，但在 3.x 系列中不會被移除。

(8) 參🄵 **PEP 237**。

備🄵

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

也參考

PEP 461 - Adding % formatting to bytes and bytearray

在 3.5 版被加入.

4.10.5 Memory Views

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

class memoryview(*object*)

Create a *memoryview* that references *object*. *object* must support the buffer protocol. Built-in objects that support the buffer protocol include *bytes* and *bytearray*.

A *memoryview* has the notion of an *element*, which is the atomic memory unit handled by the originating *object*. For many simple types such as *bytes* and *bytearray*, an element is a single byte, but other types such as *array.array* may have bigger elements.

`len(view)` is equal to the length of *tolist*, which is the nested list representation of the view. If `view.ndim = 1`, this is equal to the number of elements in the view.

在 3.12 版的變更: If `view.ndim == 0`, `len(view)` now raises *TypeError* instead of returning 1.

The *itemsize* attribute will give you the number of bytes in a single element.

A *memoryview* supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If *format* is one of the native format specifiers from the *struct* module, indexing with an integer or a tuple of integers is also supported and returns a single *element* with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed with tuples of exactly *ndim* integers where *ndim* is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

One-dimensional memoryviews of *hashable* (read-only) types with formats 'B', 'b' or 'c' are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

在 3.3 版的變更: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats 'B', 'b' or 'c' are now *hashable*.

在 3.4 版的變更: memoryview is now registered automatically with *collections.abc.Sequence*

在 3.5 版的變更: memoryviews can now be indexed with tuple of integers.

在 3.14 版的變更: memoryview is now a *generic type*.

memoryview 有幾個方法:

__eq__(*exporter*)

A memoryview and a **PEP 3118** exporter are equal if their shapes are equivalent and if all

corresponding values are equal when the operands' respective format codes are interpreted using *struct* syntax.

For the subset of *struct* format strings currently supported by *tolist()*, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the *struct* module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating-point numbers, `v is w` does *not* imply `v == w` for memoryview objects.

在 3.3 版的變更: Previous versions compared the raw memory disregarding the item format and the logical array structure.

tobytes(*order='C'*)

Return the data in the buffer as a bytestring. This is equivalent to calling the *bytes* constructor on the memoryview.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. *tobytes()* supports all format strings, including those that are not in *struct* module syntax.

在 3.8 版被加入: *order* can be {'C', 'F', 'A'}. When *order* is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. *order=None* is the same as *order='C'*.

hex(*, *bytes_per_sep=1*)

hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

在 3.5 版被加入.

在 3.8 版的變更: Similar to *bytes.hex()*, *memoryview.hex()* now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

tolist()

Return the data in the buffer as a list of elements.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

在 3.3 版的變更: *tolist()* now supports all single character native formats in *struct* module syntax as well as multi-dimensional representations.

toreadonly()

Return a readonly version of the memoryview object. The original memoryview object is unchanged.

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

在 3.8 版被加入.

release()

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a *bytearray* would temporarily forbid resizing); therefore, calling release() is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a *ValueError* (except *release()* itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

在 3.2 版被加入.

cast(*format*, */*)

cast(*format*, *shape*, */*)

Cast a memoryview to a new format or shape. *shape* defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-*contiguous* and C-contiguous -> 1D.

The destination format is restricted to a single element native format in *struct* syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length. Note that all byte lengths may depend on the operating system.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Cast 1D/unsigned bytes to 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
```

(繼續下一頁)

(繼續上一頁)

```
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

在 3.3 版被加入.

在 3.5 版的變更: The source format is no longer restricted when casting to a byte view.

count(*value*, /)

計算 *value* 的出現次數

在 3.14 版被加入.

index(*value*, *start=0*, *stop=sys.maxsize*, /)

Return the index of the first occurrence of *value* (at or after index *start* and before index *stop*).

在找不到 *value* 時引發`ValueError`。

在 3.14 版被加入.

There are also several readonly attributes available:

obj

memoryview 的底層物件:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

在 3.3 版被加入.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

在 3.3 版被加入.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in *struct* module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. *tolist()*) are restricted to native single element formats.

在 3.3 版的變更: format `'B'` is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape

A tuple of integers the length of *ndim* giving the shape of the memory as an N-dimensional array.

在 3.3 版的變更: An empty tuple instead of `None` when ndim = 0.

strides

A tuple of integers the length of *ndim* giving the size in bytes to access each element for each dimension of the array.

在 3.3 版的變更: An empty tuple instead of `None` when ndim = 0.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-*contiguous*.

在 3.3 版被加入.

f_contiguous

A bool indicating whether the memory is Fortran *contiguous*.

在 3.3 版被加入.

contiguous

A bool indicating whether the memory is *contiguous*.

在 3.3 版被加入.

4.11 Set Types --- `set`, `frozenset`

A *set* object is an unordered collection of distinct *hashable* objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in *dict*, *list*, and *tuple* classes, and the *collections* module.)

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, *set* and *frozenset*. The *set* type is mutable --- the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The *frozenset* type is immutable and *hashable* --- its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the *set* constructor.

The constructors for both classes work the same:

class set(*iterable=()*, */*)

class frozenset(*iterable=()*, */*)

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be *hashable*. To represent sets of sets, the inner sets must be *frozenset* objects. If *iterable* is not specified, a new empty set is returned.

集合可以以多種方式建立:

- Use a comma-separated list of elements within braces: `{'jack', 'sjoerd'}`
- Use a set comprehension: `{c for c in 'abracadabra' if c not in 'abc'}`
- Use the type constructor: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

Instances of *set* and *frozenset* provide the following operations:

len(s)

Return the number of elements in set *s* (cardinality of *s*).

x in s

Test *x* for membership in *s*.

x not in s

Test *x* for non-membership in *s*.

isdisjoint(*other*, /)

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

issubset(*other*, /)

set <= other

Test whether every element in the set is in *other*.

set < other

Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

issuperset(*other*, /)

set >= other

Test whether every element in *other* is in the set.

set > other

Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

union(**others*)

set | other | ...

Return a new set with elements from the set and all others.

intersection(**others*)

set & other & ...

Return a new set with elements common to the set and all others.

difference(**others*)

set - other - ...

Return a new set with elements in the set that are not in the others.

symmetric_difference(*other*, /)

set ^ other

Return a new set with elements in either the set or *other* but not both.

copy()

Return a shallow copy of the set.

Note, the non-operator versions of *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()*, and *issuperset()* methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both *set* and *frozenset* support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of *set* are compared to instances of *frozenset* based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`.

Since sets only define partial ordering (subset relationships), the output of the *`list.sort()`* method is undefined for lists of sets.

Set elements, like dictionary keys, must be *hashable*.

Binary operations that mix *`set`* instances with *`frozenset`* return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of *`frozenset`*.

The following table lists operations available for *`set`* that do not apply to immutable instances of *`frozenset`*:

`update`(**others*)

`set |= other | ...`

Update the set, adding elements from all others.

`intersection_update`(**others*)

`set &= other & ...`

Update the set, keeping only elements found in it and all others.

`difference_update`(**others*)

`set -= other | ...`

Update the set, removing elements found in others.

`symmetric_difference_update`(*other*, /)

`set ^= other`

Update the set, keeping only elements found in either set, but not in both.

`add`(*elem*, /)

將元素 *elem* 加入集合。

`remove`(*elem*, /)

Remove element *elem* from the set. Raises *`KeyError`* if *elem* is not contained in the set.

`discard`(*elem*, /)

如果 *elem* 存在於集合中則將其移除。

`pop`()

Remove and return an arbitrary element from the set. Raises *`KeyError`* if the set is empty.

`clear`()

從集合中移除所有元素。

Note, the non-operator versions of the *`update()`*, *`intersection_update()`*, *`difference_update()`*, and *`symmetric_difference_update()`* methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, *`remove()`*, and *`discard()`* methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.12 Mapping Types --- `dict`

A *mapping* object maps *hashable* values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in *`list`*, *`set`*, and *`tuple`* classes, and the *`collections`* module.)

A dictionary's keys are *almost* arbitrary values. Values that are not *hashable*, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Values that compare equal (such as `1`, `1.0`, and `True`) can be used interchangeably to index the same dictionary entry.

class dict(***kwargs*)

class dict(*mapping*, */*, ***kwargs*)

class dict(*iterable*, */*, ***kwargs*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

字典可以用數種方式建立：

- Use a comma-separated list of `key: value` pairs within braces: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- Use a dict comprehension: `{}`, `{x: x ** 2 for x in range(10)}`
- Use the type constructor: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it defines a `keys()` method, a dictionary is created by calling `__getitem__()` on the argument with each returned key from the method. Otherwise, the positional argument must be an *iterable* object. Each item in the iterable must itself be an iterable with exactly two elements. The first element of each item becomes a key in the new dictionary, and the second element the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs (regardless of ordering). Order comparisons ('<', '<=', '>=', '>') raise *TypeError*. To illustrate dictionary creation and equality, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

在 3.7 版的變更: Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

list(d)

Return a list of all the keys used in the dictionary *d*.

len(d)

Return the number of items in the dictionary *d*.

d[key]

Return the item of *d* with key *key*. Raises a *KeyError* if *key* is not in the map.

If a subclass of dict defines a method `__missing__()` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__()`. If `__missing__()` is not defined, *KeyError* is raised. `__missing__()` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of *collections.Counter*. A different `__missing__()` method is used by *collections.defaultdict*.

d[key] = value

將 `d[key]` 設🄵 *value*。

del d[key]

從 *d* 中移除 `d[key]`。若 *key* 不在對映中則引發*KeyError*。

key in d

若 *d* 有鍵 *key* 則回傳 `True`，否則回傳 `False`。

key not in d

等價於 `not key in d`。

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

clear()

從字典中移除所有項目。

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from *iterable* and values set to *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to `None`. All of the values refer to just a single instance, so it generally doesn't make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get(*key*, *default=None*, /)

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return a new view of the dictionary's items (`(key, value)` pairs). See the *documentation of view objects*.

keys()

Return a new view of the dictionary's keys. See the *documentation of view objects*.

pop(*key*, /)

pop(*key*, *default*, /)

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem()

Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in LIFO (last-in, first-out) order.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

在 3.7 版的變更: LIFO order is now guaranteed. In prior versions, `popitem()` would return an arbitrary key/value pair.

reversed(d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed(d.keys())`.

在 3.8 版被加入.

setdefault(*key*, *default=None*, /)

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update(***kwargs*)

update(*mapping*, /, ***kwargs*)

update(*iterable*, /, ***kwargs*)

Update the dictionary with the key/value pairs from *mapping* or *iterable* and *kwargs*, overwriting existing keys. Return `None`.

`update()` accepts either another object with a `keys()` method (in which case `__getitem__()` is called with every key returned from the method) or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

values()

Return a new view of the dictionary's values. See the *documentation of view objects*.

An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

Create a new dictionary with the merged keys and values of *d* and *other*, which must both be dictionaries. The values of *other* take priority when *d* and *other* share keys.

在 3.9 版被加入.

d |= other

Update the dictionary *d* with keys and values from *other*, which may be either a *mapping* or an *iterable* of key/value pairs. The values of *other* take priority when *d* and *other* share keys.

在 3.9 版被加入.

Dictionaries and dictionary views are reversible.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

在 3.8 版的變更: Dictionaries are now reversible.

也參考

types.MappingProxyType can be used to create a read-only view of a *dict*.

4.12.1 字典視圖物件

The objects returned by *dict.keys()*, *dict.values()* and *dict.items()* are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

len(dictview)

Return the number of entries in the dictionary.

iter(dictview)

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using *zip()*: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a *RuntimeError* or fail to iterate over all entries.

在 3.7 版的變更: Dictionary order is guaranteed to be insertion order.

x in dictview

Return `True` if *x* is in the underlying dictionary's keys, values or items (in the latter case, *x* should be a `(key, value)` tuple).

reversed(dictview)

Return a reverse iterator over the keys, values or items of the dictionary. The view will be iterated in reverse order of the insertion.

在 3.8 版的變更: Dictionary views are now reversible.

dictview.mapping

Return a *types.MappingProxyType* that wraps the original dictionary to which the view refers.

在 3.10 版被加入.

Keys views are set-like since their entries are unique and *hashable*. Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class *collections.abc.Set* are available (for example, ==, <, or ^). While using set operators, set-like views accept any iterable as the other operand, unlike sets which only accept sets as the input.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

4.13 情境管理器型🄵

Python 的 `with` 陳述式支援了由情境管理器定義之 runtime 情境的概念，要使用兩個方法來實作，該方法讓使用者定義類🄵能🄵去定義 runtime 情境，且該情境在執行陳述式主體 (statement body) 之前進入、在陳述式結束時退出：

`contextmanager.__enter__()`

輸入 runtime 情境🄵回傳此物件或者與 runtime 情境相關的另一個物件。此方法回傳的值有被綁定到使用此情境管理器的 `with` 陳述式的 `as` 子句中的識🄵字。

一個會回傳自己的情境管理器範例是*file object*。檔案物件從 __enter__() 回傳自己，以允許將*open()* 用作 `with` 陳述式中的情境運算式。

一個會回傳相關物件的情境管理器範例是由*decimal.localcontext()* 回傳的管理器。這些管理器將有效的十進位情境設定🄵原始十進位情境的副本，然後回傳該副本。這允許對 `with` 陳述式主

體中的目前十進位情境進行更改，而不會影響 `with` 陳述式外部的程式碼。

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)

退出 runtime 情境🄴回傳布林旗標以表示是否應抑制曾發生的任何例外。如果在執行 `with` 陳述式主體時發生例外，則引數包含例外型🄴、值和回溯 (traceback) 資訊。否則，所有三個引數都是 `None`。

從此方法回傳 true 值將導致 `with` 陳述式抑制例外🄴繼續執行緊接著 `with` 陳述式之後的陳述式。否則，該例外將在該方法執行完畢後繼續傳播 (propagate)。執行此方法期間發生的例外會取代 `with` 陳述式主體中發生的任何例外。

傳入的例外不應明確重新引發 - 取而代之的是，此方法應回傳 false 值以指示該方法已成功完成且不希望抑制引發的例外。這讓情境管理程式碼能輕松檢測 `__exit__()` 方法是否曾實際失敗過。

Python 定義了多個情境管理器來支援簡單的執行緒同步、檔案或其他物件的提示關閉以及對有效十進位算術情境的更簡單操作。除了情境管理協定的實作之外，不會對特定型🄵進行特殊處理。更多範例請參🄵*contextlib* 模組。

Python 的*generator* 和*contextlib.contextmanager* 裝飾器提供了一種便捷的方法來實作這些協定。如果🄵生器函式以*contextlib.contextmanager* 裝飾器裝飾，它將回傳一個有實作出需要的*__enter__()* 和*__exit__()* 方法的情境管理器，而不是由未裝飾🄵生器函式🄵生的🄵代器。

請注意，Python/C API 中 Python 物件的型🄵結構中的任何方法都🄵有特定的槽。想要定義這些方法的擴充型🄵必須將它們作🄵普通的 Python 可存取方法提供。與設定 runtime 情境的開銷相比，單一類🄵字典查找的開銷可以忽略不計。

4.14 型🄵🄵釋的型🄵 --- 泛型🄵名 (Generic Alias)、聯合 (Union)

型🄵🄵釋 的核心🄵建型🄵是泛型🄵名和聯合。

4.14.1 泛型🄵名型🄵

`GenericAlias` 物件通常是透過下標 (subscripting) 一個類🄵來建立的。它們最常與容器類🄵一起使用，像是*list* 或*dict*。例如 `list[int]` 是一個 `GenericAlias` 物件，它是透過使用引數*int* 來下標 `list` 類🄵而建立的。`GenericAlias` 物件主要會與型🄵🄵釋 一起使用。

> **備🄵**
>
> 通常只有當類🄵有實作特殊方法 `__class_getitem__()` 時才可以去下標該類🄵。

將一個 `GenericAlias` 物件用作*generic type* 的代理，實作參數化泛型 *(parameterized generics)*。

對於一個容器類🄵，提供給該類🄵的下標引數可以代表物件所包含元素的型🄵。例如 `set[bytes]` 可以用於型🄵🄵釋來表示一個*set*，其中所有元素的型🄴都是*bytes*。

對於定義 `__class_getitem__()` 但不是容器的類🄵，提供給該類🄵的下標引數通常會指示物件上有定義的一個或多個方法的回傳型🄵。例如正規表示式可以用於*str* 和*bytes* 資料型🄵：

- 如果 `x = re.search('foo', 'foo')`，`x` 將會是一個*re.Match* 物件，其中 `x.group(0)` 和 `x[0]` 的回傳值都是*str* 型🄵。我們就可以用 `GenericAlias` `re.Match[str]` 在型🄵🄵釋中表示這種物件。
- 如果 `y = re.search(b'bar', b'bar')`（注意*bytes* 的 `b`），`y` 也會是 `re.Match` 的實例，但 `y.group(0)` 和 `y[0]` 的回傳值的型🄵都是*bytes*。在型🄵🄵釋中，我們將用 `re.Match[bytes]` 來表示各種*re.Match* 物件。

`GenericAlias` 物件是*types.GenericAlias* 類🄵的實例，也可以用來直接建立 `GenericAlias` 物件。

`T[X, Y, ...]`

建立一個 `GenericAlias` 來表示一個型🄵 `T`，其以型🄵 *X*、*Y* 等（取🄵於所使用的 `T`）來參數化。例如，一個函式需要一個包含*float* 元素的*list*：

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

對映物件的另一個範例，使用*dict*，它是一個泛型型𝔽，需要兩個型𝔽參數，分𝔽表示鍵型𝔽和值型𝔽。在此範例中，函式需要一個 dict，其帶有*str* 型𝔽的鍵和*int* 型𝔽的值：

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

𝔽建函式*isinstance()* 和*issubclass()* 不接受 GenericAlias 型𝔽作𝔽第二個引數：

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

Python runtime 不𝔽制執行型𝔽𝔽釋。這也擴展到泛型型𝔽及其型𝔽參數。當從 GenericAlias 建立容器物件時，不會檢查容器中元素的型𝔽。例如，不鼓勵使用以下程式碼，但 runtime 不會出現錯誤：

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

此外，參數化泛型在物件建立期間會擦除 (erase) 型𝔽參數：

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

在泛型上呼叫*repr()* 或*str()* 會顯示參數化型𝔽：

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

𝔽防止像是 dict[str][str] 的錯誤出現，泛型容器的 __getitem__() 方法會在這種情𝔽下引發例外：

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

然而當使用型𝔽變數 *(type variable)* 時，此類運算式是有效的。索引的元素數量必須與 GenericAlias 物件的*__args__* 中的型𝔽變數項目一樣多：

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

標準泛型類𝔽

以下標准函式庫類𝔽有支援參數化泛型。此列表𝔽非詳盡無遺。

- *tuple*
- *list*
- *dict*
- *set*

- *frozenset*
- *type*
- *asyncio.Future*
- *asyncio.Task*
- *collections.deque*
- *collections.defaultdict*
- *collections.OrderedDict*
- *collections.Counter*
- *collections.ChainMap*
- *collections.abc.Awaitable*
- *collections.abc.Coroutine*
- *collections.abc.AsyncIterable*
- *collections.abc.AsyncIterator*
- *collections.abc.AsyncGenerator*
- *collections.abc.Iterable*
- *collections.abc.Iterator*
- *collections.abc.Generator*
- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*
- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*
- *collections.abc.ValuesView*
- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*

- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

GenericAlias 物件的特殊屬性

所有參數化泛型都有實作特殊的唯讀屬性。

genericalias.**__origin__**

此屬性指向非參數化泛型類𝔽:

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.**__args__**

此屬性是傳遞給泛型類𝔽之原始 `__class_getitem__()` 的泛型型𝔽*tuple*（長度可以𝔽 1）:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**

此屬性是個會被延遲計算 (lazily computed) 的元組（可能𝔽空），包含了在 `__args__` 中找得到的不重𝔽型𝔽變數:

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

備𝔽

具有*typing.ParamSpec* 參數的一個 `GenericAlias` 物件在替𝔽後可能𝔽有正確的 `__parameters__`，因𝔽*typing.ParamSpec* 主要用於𝔽態型𝔽檢查。

genericalias.**__unpacked__**

如果𝔽名已使用 `*` 運算子解包 (unpack) 則𝔽 true 的布林值（請參𝔽*TypeVarTuple*）。

在 3.11 版被加入.

也參考

PEP 484 - 型🄴提示
引入 Python 的型🄵🄴釋框架。

PEP 585 - 標准集合 (Standard Collections) 中的型🄴提示泛型
引入原生參數化標准函式庫類🄴的能力，前提是它們有實作特殊的類🄴方法 `__class_getitem__()`。

泛型、使用者定義泛型和*typing.Generic*
有關如何實作可以在 runtime 參數化🄴能被🄴態型🄴檢查器理解的泛型類🄴的文件。

在 3.9 版被加入.

4.14.2 聯合型🄵 (Union Type)

一個聯合物件可以保存多個型🄴物件 *(type object)*之 `|`（位元 or）運算的值。這些型🄴主要用於型🄵🄴釋 *(type annotation)*。與下標*typing.Union* 相比，聯合型🄴運算式可以讓型🄴提示語法更清晰簡潔。

X | Y | ...

定義一個包含 *X*、*Y* 等型🄴的聯合物件。`X | Y` 表示 X 或 Y。它相當於 `typing.Union[X, Y]`。舉例來🄴，下列函式需要一個型🄵🄴*int* 或*float* 的引數：

```
def square(number: int | float) -> int | float:
    return number ** 2
```

備🄴

不能在 runtime 使用 `|` 運算元 (operand) 來定義有一個以上的成員🄴向前參照 (forward reference) 的聯合。例如 `int | "Foo"`，其中 `"Foo"` 是對未定義類🄴的參照，將在 runtime 失敗。對於包含向前參照的聯合，請將整個運算式以字串呈現，例如 `"int | Foo"`。

union_object == other

聯合物件可以與其他聯合物件一起進行相等性測試。細節如下：

- 聯合的聯合會被扁平化：

```
(int | str) | float == int | str | float
```

- 冗余型🄴會被🄴除：

```
int | str | int == int | str
```

- 比較聯合時，順序會被忽略：

```
int | str == str | int
```

- 它會建立*types.UnionType* 的實例：

```
int | str == typing.Union[int, str]
type(int | str) is typing.Union
```

- 可選型🄴可以表示🄴與 `None` 的聯合：

```
str | None == typing.Optional[str]
```

isinstance(obj, union_object)

issubclass(obj, union_object)

聯合物件也支援*isinstance()* 和*issubclass()* 的呼叫:

```
>>> isinstance("", int | str)
True
```

然而聯合物件中的參數化泛型則無法被檢查:

```
>>> isinstance(1, int | list[int])  # short-circuit evaluation
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

構成聯合物件的對使用者公開型F (user-exposed type) 可以透過*types.UnionType* 存取F用於*isinstance()* 檢查:

```
>>> import typing
>>> isinstance(int | str, typing.Union)
True
>>> typing.Union()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'typing.Union' instances
```

備F

新增了型F物件的 `__or__()` 方法來支援 `X | Y` 語法。如果元類F有實作 `__or__()`，則 Union 可以覆寫 (override) 它:

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

也參考

PEP 604 -- PEP 提出 `X | Y` 語法和聯合型F。

在 3.10 版被加入.

在 3.14 版的變更: Union objects are now instances of *typing.Union*. Previously, they were instances of *types.UnionType*, which remains an alias for *typing.Union*.

4.15 其他F建型F

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

4.15.1 模組

The only special operation on a module is attribute access: `m.name`, where *m* is a module and *name* accesses a name defined in *m*'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named *foo* to exist, rather it requires an (external) *definition* for a module named *foo* somewhere.)

A special attribute of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be `1`, but you can't write `m.__dict__ = {}`). Modifying `__dict__` directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.15.2 類🄵與類🄵實例

請見 objects 和 class。

4.15.3 函式

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

更多資訊請見 function。

4.15.4 方法

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as *append()* on lists) and class instance method. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`method.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an *AttributeError* being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # 不得設定於方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

更多資訊請見 instance-methods。

4.15.5 程式碼物件

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in *compile()* function and can be extracted from function objects through their `__code__` attribute. See also the *code* module.

存取 `__code__` 會引發一個附帶引數 `obj` 與 `"__code__"` 的稽核事件 `object.__getattr__`。

A code object can be executed or evaluated by passing it (instead of a source string) to the *exec()* or *eval()* built-in functions.

更多資訊請見 types。

4.15.6 Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in function *type()*. There are no special operations on types. The standard module *types* defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

4.15.7 Null 物件

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

它被寫🄵 `None`。

4.15.8 Ellipsis 物件

This object is commonly used used to indicate that something is omitted. It supports no special operations. There is exactly one ellipsis object, named *Ellipsis* (a built-in name). `type(Ellipsis)()` produces the *Ellipsis* singleton.

它被寫🄵 `Ellipsis` 或 `...`。

In typical use, `...` as the `Ellipsis` object appears in a few different places, for instance:

- In type annotations, such as *callable arguments* or *tuple elements*.
- As the body of a function instead of a pass statement.
- In third-party libraries, such as Numpy's slicing and striding.

Python also uses three dots in ways that are not `Ellipsis` objects, for instance:

- Doctest's *ELLIPSIS*, as a pattern for missing content.
- The default Python prompt of the *interactive* shell when partial input is incomplete.

Lastly, the Python documentation often uses three dots in conventional English usage to mean omitted content, even in code examples that also use them as the `Ellipsis`.

4.15.9 NotImplemented 物件

This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See comparisons for more information. There is exactly one *NotImplemented* object. `type(NotImplemented)()` produces the singleton instance.

它被寫🄵 `NotImplemented`。

4.15.10 🄵部物件

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.16 特殊屬性

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the *dir()* built-in function.

`definition.`**`__name__`**

The name of the class, function, method, descriptor, or generator instance.

`definition.`**`__qualname__`**

The *qualified name* of the class, function, method, descriptor, or generator instance.

在 3.3 版被加入.

`definition.`**`__module__`**

The name of the module in which a class or function was defined.

`definition.`**`__doc__`**

The documentation string of a class or function, or `None` if undefined.

`definition.`**`__type_params__`**

The type parameters of generic classes, functions, and *type aliases*. For classes and functions that are not generic, this will be an empty tuple.

在 3.12 版被加入.

4.17 Integer string conversion length limitation

CPython has a global limit for converting between *int* and *str* to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The *int* type in CPython is an arbitrary length number stored in binary form (commonly known as a "bignum"). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid **CVE 2020-10735**.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a *ValueError* is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion: value has 5432␣
↪digits; use sys.set_int_max_str_digits() to increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion; use sys.set_int_
↪max_str_digits() to increase the limit
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in *sys.int_info.default_max_str_digits*. The lowest limit that can be configured is 640 digits as provided in *sys.int_info.str_digits_check_threshold*.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

在 3.11 版被加入.

4.17.1 受影響的 API

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` 以預設的 10 🄵底。
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`。
- `repr(integer)`。
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *int.from_bytes()* 和*int.to_bytes()*。
- *hex()*、*oct()*、*bin()*。
- *格式規格 (Format Specification)* 迷你語言 for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.17.2 設定限制

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, 例如 `python3 -X int_max_str_digits=640`
- *sys.flags.int_max_str_digits* contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of *sys.int_info.default_max_str_digits* was used during initialization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- *sys.get_int_max_str_digits()* and *sys.set_int_max_str_digits()* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *sys.int_info*:

- *sys.int_info.default_max_str_digits* is the compiled-in default limit.
- *sys.int_info.str_digits_check_threshold* is the lowest accepted value for the limit (other than 0 which disables it).

在 3.11 版被加入.

警示

Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.17.3 建議的配置

The default *sys.int_info.default_max_str_digits* is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.12.

範例:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

CHAPTER 5

內建的例外

在 Python 中，所有例外必須是從*BaseException* 衍生的類別的實例。在陳述式 try 搭配 except 子句別提到一個特定的類別時，那個子句也會處理任何從該類別衍生的例外類別（但不會處理該類別衍生自的例外類別）。兩個不是由子類別關別關聯起來的例外類別永遠不相等，就算它們有相同的名稱也是如此。

此章節別列出的別建例外可以從直譯器或別建函式別生。除了特別提到的地方之外，它們會有一個關聯值表示錯誤發生的詳細原因。這可能是一個字串，或者是一些資訊項目組成的元組（例如一個錯誤代碼及一個解釋該代碼的字串）。這個關聯值通常當作引數傳遞給例外類別的建構函式。

使用者的程式碼可以引發別建例外。這可以用來測試例外處理器或者用來回報一個錯誤條件，就像直譯器會引發相同例外的情別；但需要注意的是別有任何方式可以避免使用者的程式碼引發不適當的錯誤。

可以從別建的例外類別定義新的例外子類別；程式設計師被鼓勵從*Exception* 類別或其子類別衍生新的例外，而不是從*BaseException* 來衍生。更多關於定義例外的資訊可以在 Python 教學中的 tut-userexceptions 別取得。

5.1 例外的情境

三個例外物件上的屬性提供關於引發此例外的情境的資訊：

BaseException.**__context__**

BaseException.**__cause__**

BaseException.**__suppress_context__**

當引發一個新的例外而同時有另一個例外已經正在被處理時，這個新例外的 __context__ 屬性會自動被設成那個已處理的例外。當使用 except 或 finally 子句或 with 陳述式的時候例外會被處理。

這個隱含的例外情境可以透過使用 from 搭配 raise 來補充明確的原因：

```
raise new_exc from original_exc
```

在 from 後面的運算式必須是一個例外或 None。它將會被設定成所引發例外的 __cause__。設定 __cause__ 也隱含地設定 __suppress_context__ 屬性別 True，因此使用 raise new_exc from None 實際上會以新的例外取代舊的例外以利於顯示（例如轉別*KeyError* 別*AttributeError*），同時保持舊的例外可以透過 __context__ 取得以方便 debug 的時候檢查。

預設的回溯 (traceback) 顯示程式碼會顯示這些連鎖的例外 (chained exception) 加上例外本身的回溯。當存在的時候，在 __cause__ 中明確地連鎖的例外總是會被顯示。而在 __context__ 中隱含地連鎖的例外只有當 __cause__ 是*None* 且 __suppress_context__ 是 false 時才會顯示。

在任一種情🄵下，例外本身總是會顯示在任何連鎖例外的後面，因此回溯的最後一列總是顯示最後一個被引發的例外。

5.2 繼承自🄵建的例外

使用者的程式碼可以建立繼承自例外型🄵的子類🄵。建議一次只繼承一種例外型🄵以避免在基底類🄵之間如何處理 `args` 屬性的任何可能衝突，以及可能的記憶體🄵局 (memory layout) 不相容。

🄵了效率，大部分的🄵建例外使用 C 來實作，參考 Objects/exceptions.c。一些例外有客🄵化的記憶體🄵局，使其不可能建立一個繼承多種例外型🄵的子類🄵。型🄵的記憶體🄵局是實作細節且可能會在不同 Python 版本間改變，造成未來新的衝突。因此，總之建議避免繼承多種例外型🄵。

5.3 基底類🄵 (base classes)

以下的例外大部分被用在當作其他例外的基底類🄵。

exception BaseException

所有🄵建例外的基底類🄵。這不是🄵了讓使用者定義的類🄵直接繼承（可以使用*Exception*）。如果在這個類🄵的實例上呼叫*str()*，會回傳實例的引數的表示，或者🄵有引數的時候會回傳空字串。

args

提供給該例外建構函式的引數元組。一些🄵建的例外（像是*OSError*）預期接受特定數量的引數🄵賦予該元組的每一個元素一個特🄵的意義，其他例外則通常用一個提供錯誤訊息的單一字串來呼叫。

with_traceback(*tb*)

此方法設定 *tb* 🄵該例外的新的回溯🄵回傳該例外物件。在 **PEP 3134** 的例外連鎖功能變得可用之前，此方法曾被更普遍使用。下面的範例顯示我們如何將 `SomeException` 的實例轉🄵🄵 `OtherException` 的實例同時保留回溯。一旦被引發，目前的 frame 會被加進 `OtherException` 的回溯，就像原來 `SomeException` 的回溯會發生的一樣，我們允許它被傳遞給呼叫者：

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

可寫入的欄位，儲存關聯到該例外的回溯物件。也可以參考 raise。

add_note(*note*)

新增字串 `note` 到例外的備🄵，在標準的回溯🄵，備🄵出現在例外字串的後面。如果 `note` 不是字串則引發*TypeError*。

在 3.11 版被加入.

__notes__

該例外的備🄵串列，使用*add_note()* 來新增。此屬性在*add_note()* 被呼叫的時候建立。

在 3.11 版被加入.

exception Exception

所有🄵建、非系統退出 (non-system-exiting) 的例外都衍生自此類🄵。所有使用者定義的例外應該也要衍生自此類🄵。

exception ArithmeticError

各種運算錯誤所引發的那些🄵建例外：*OverflowError*、*ZeroDivisionError*、*FloatingPointError* 的基底類🄵。

exception BufferError

當緩衝 (buffer) 相關的操作無法被執行時會引發此例外。

exception LookupError

當使用在對映或序列上的鍵或索引是無效的時候所引發的例外：*IndexError*、*KeyError* 的基底類🄵。這可以被*codecs.lookup()* 直接引發。

5.4 實體例外

以下的例外是通常會被引發的例外。

exception AssertionError

當 `assert` 陳述式失敗的時候被引發。

exception AttributeError

當屬性參照（參考 attribute-references）或賦值失敗的時候被引發。（當物件根本不支援屬性參照或屬性賦值的時候，*TypeError* 會被引發。）

可選的僅限關鍵字引數 *name* 和 *obj* 會設定對應的屬性：

name

嘗試被存取的屬性名稱。

obj

存取指定屬性的物件。

在 3.10 版的變更: 新增*name* 與*obj* 屬性。

exception EOFError

當*input()* 函式在🄵有讀到任何資料而到達檔案結尾 (end-of-file, EOF) 條件的時候被引發。（注意：`io.IOBase.read()` 和*io.IOBase.readline()* 方法當達到 EOF 時會回傳空字串。）

exception FloatingPointError

目前🄵有被使用。

exception GeneratorExit

當*generator* 或*coroutine* 被關閉的時候被引發；參考 `generator.close()` 和 `coroutine.close()`。此例外直接繼承自*BaseException* 而不是*Exception*，因🄵技術上來🄵這不是一個錯誤。

exception ImportError

當 `import` 陳述式嘗試載入模組遇到問題的時候會被引發。當 `from ... import` 🄵的"from list" 包含找不到的名稱時也會被引發。

可選的僅限關鍵字引數 *name* 和 *path* 設定對應的屬性：

name

嘗試引入 (import) 的模組名稱。

path

觸發此例外的任何檔案的路徑。

在 3.3 版的變更: 新增*name* 與*path* 屬性。

exception ModuleNotFoundError

ImportError 的子類🄵，當模組不能被定位的時候會被 `import` 所引發。當在*sys.modules* 🄵找到 `None` 時也會被引發。

在 3.6 版被加入.

exception IndexError

當序列的索引超出範圍的時候會被引發。（切片索引 (slice indices) 會默默地被截短使其能落在允許的範圍🄵；如果索引不是整數，*TypeError* 會被引發。）

exception KeyError

當對映（字典）的鍵無法在已存在的鍵的集合中被找到時會被引發。

exception KeyboardInterrupt

當使用者輸入中斷鍵 (interrupt key)（一般來🄵是 `Control-C` 或 `Delete`）時會被引發。在執行過程中，會定期檢查是否🄵生中斷。此例外繼承自*BaseException* 以防止意外地被捕捉*Exception* 的程式碼所捕捉，而因此讓直譯器無法結束。

> **備🄵**
>
> 捕捉*KeyboardInterrupt* 需要特殊的考量。因🄵它可以在無法預期的時間點被引發，可能在某些情🄵下讓正在跑的程式處在一個不一致的狀態。一般來🄵最好讓*KeyboardInterrupt* 越快結束程式越好，或者完全避免引發它。（參考*訊號處理程式與例外的🄵明*。）

exception MemoryError

當一個操作用光了記憶體但情🄵還可能被修復 (rescued)（透過🄵除一些物件）的時候被引發。關聯值是一個字串，表示什🄵類型的（🄵部）操作用光了記憶體。需注意的是因🄵底層的記憶體管理架構（C 的 `malloc()` 函式），直譯器可能無法總是完整地從該情🄵中修復；僅管如此，它還是引發例外以讓堆🄵回溯可以被印出，以防原因出在失控的程式。

exception NameError

當找不到本地或全域的名稱時會被引發。這只應用在不合格的名稱 (unqualified name) 上。關聯值是一個錯誤訊息，包含那個無法被找到的名稱。

可選的僅限關鍵字引數 *name* 設定對應的屬性：

name

嘗試被存取的變數名稱。

在 3.10 版的變更: 新增*name* 屬性。

exception NotImplementedError

此例外衍生自*RuntimeError*。在使用者定義的基礎類🄵🄵，當抽象方法要求衍生類🄵覆寫該方法時應該要引發此例外，或者當類🄵正在開發中，可用此例外表示還需要加入真正的實作。

> **備🄵**
>
> 此例外不應該用來表示根本🄵有要支援的運算子或方法 —— 在這個情🄵下可以讓該運算子或方法保持未定義，或者如果是子類🄵的話將其設成*None*。

> **警示**
>
> `NotImplementedError` 和 `NotImplemented` 🄵不是可互🄵的。這個例外只能用在上面描述的情🄵；參考*NotImplemented* 🄵關於🄵建常數正確使用方法的細節。

exception OSError(*[arg]*)

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

當系統函式回傳系統相關錯誤，包含像" 找不到檔案" 或" 硬碟已滿" 的 I/O 失敗會引發此例外（而非不合法的引數或其他次要的錯誤）。

建構函式的第二種形式會設定以下描述的相對應屬性。如果🄵有給定則屬性預設🄵*None*。🄵了向後相容，如果傳入三個引數，*args* 屬性只會是包含建構函式前兩個引數的雙元素元組。

如同下面的*作業系統例外*所描述，實際上建構函式通常回傳*OSError* 的子類🄵。會依據最後*errno* 的值🄵定特定子類🄵。這個行🄵只發生在直接建構*OSError* 或透過🄵名，且🄵生子類🄵的時候不會被繼承。

errno

從 C 變數 `errno` 而來的數值錯誤代碼。

winerror

在 Windows 下，這會提供你原生的 Windows 錯誤代碼。而*errno* 屬性是一個該原生錯誤代碼對於 POSIX 來🄵的近似翻譯。

在 Windows 下，如果建構函式引數 *winerror* 是整數，則*errno* 屬性會根據該 Windows 錯誤代碼來🄵定，且 *errno* 引數會被忽略。在其他平台上，*winerror* 引數會被忽略，而*winerror* 屬性會不存在。

strerror

作業系統提供的對應錯誤訊息。在 POSIX 下會使用 C 函式 `perror()` 做格式化，而在 Windows 下會使用 `FormatMessage()`。

filename

filename2

對於包含檔案系統路徑的例外（像是*open()* 或*os.unlink()*），*filename* 是傳入函式的檔案名稱。對於包含兩個檔案系統路徑的函式（像是*os.rename()*），*filename2* 對應到傳入函式的第二個檔案名稱。

在 3.3 版的變更: *EnvironmentError*、*IOError*、*WindowsError*、*socket.error*、*select.error* 及 `mmap.error` 已合🄵進*OSError*，而建構函式可能會回傳子類🄵。

在 3.4 版的變更: *filename* 屬性現在是傳入函式的原始檔名，而不是從檔案系統編碼和錯誤處理函式編碼或解碼過的名稱。🄵且新增 *filename2* 建構函式引數與屬性。

exception OverflowError

當運算操作的結果太大而無法表示的時候會引發此例外。這不會發生在整數上（會改成引發*MemoryError* 而不是放🄵）。然而，因🄵一些歷史因素，OverflowError 有時候會因🄵整數在要求範圍之外而引發。因🄵在 C 🄵面缺乏浮點數例外處理的標準化，大部分的浮點數運算都🄵有被檢查。

exception PythonFinalizationError

此例外衍生自*RuntimeError*。當一個操作在直譯器關閉（也稱作*Python 最終化 (Python finalization)*）期間被阻塞會引發此例外。

在 Python 最終化期間，能🄵以*PythonFinalizationError* 被阻塞的操作範例：

- 建立新的 Python 執行緒。
- *Joining* a running daemon thread.
- *os.fork()*。

也可以參🄵*sys.is_finalizing()* 函式。

在 3.13 版被加入: 在之前，會引發一般的*RuntimeError*。

在 3.14 版的變更: *threading.Thread.join()* can now raise this exception.

exception RecursionError

此例外衍生自*RuntimeError*。當直譯器偵測到超過最大的遞🄵深度（參考*sys.getrecursionlimit()*）時會引發此例外。

在 3.5 版被加入: 在之前，會引發一般的*RuntimeError*。

exception ReferenceError

當一個被*weakref.proxy()* 函式建立的弱參照代理 (weak reference proxy) 被用來存取已經被垃圾回收 (garbage collected) 的參照物屬性時會引發此例外。更多關於弱參照的資訊參考*weakref* 模組。

exception RuntimeError

當偵測到一個不屬於任何其他種類的錯誤時會引發此例外。關聯值是一個表示確切什🄵地方出錯的字串。

exception StopIteration

會被🄵建函式*next()* 及*iterator* 的*__next__()* 方法引發，用來表示🄵代器🄵有更多項目可以🄵生。

value

此例外物件有單一屬性 `value`，當建構此例外時會以引數給定，預設🄵*None*。

當*generator* 或*coroutine* 函式回傳時，新的*StopIteration* 實例會被引發，而該函式的回傳值會被用來當作此例外建構函式的*value* 參數。

如果🄵生器程式直接或間接引發*StopIteration*，則其會被轉🄵成*RuntimeError*（保留*StopIteration* 作🄵新例外的成因）。

在 3.3 版的變更: 新增 `value` 屬性且🄵生器函式可以用它來回傳值。

在 3.5 版的變更: 透過 `from __future__ import generator_stop` 引入 RuntimeError 的轉🄵，參考 **PEP 479**。

在 3.7 版的變更: 預設對所有程式🄵用 **PEP 479**：在🄵生器引發的*StopIteration* 錯誤會轉🄵成*RuntimeError*。

exception StopAsyncIteration

此例外必須被*asynchronous iterator* 物件的 `__anext__()` 方法引發來停止🄵代。

在 3.5 版被加入.

exception SyntaxError(*message*, *details*)

當剖析器遇到語法錯誤時會引發此例外。這可能發生在 `import` 陳述式、在呼叫🄵建函式*compile()*、*exec()* 或*eval()* 的時候，或者在讀取初始🄵本或標準輸入（也包含互動式）的時候。

例外實例的*str()* 只回傳錯誤訊息。Details 是個元組，其成員也能🄵以分開的屬性取得。

filename

發生語法錯誤所在的檔案名稱。

lineno

發生錯誤所在檔案的列號。這是以 1 開始的索引：檔案第一列的 `lineno` 是 1。

offset

發生錯誤所在該列的欄號 (column)。這是以 1 開始的索引：該列第一個字元的 `offset` 是 1。

text

涉及該錯誤的原始程式碼文字。

end_lineno

發生錯誤所在檔案的結束列號。這是以 1 開始的索引：檔案第一列的 `lineno` 是 1。

end_offset

發生錯誤所在該結束列的欄號。這是以 1 開始的索引：該列第一個字元的 `offset` 是 1。

對於發生在 f-string 欄位的錯誤，訊息會以"f-string: " 🄵前綴，而偏移量 (offset) 是從替代表達式建構的文字的偏移量。例如編譯 f'Bad {a b} field' 會得到這個 args 屬性：('f-string: ...', (", 1, 2, '(a b)n', 1, 5))。

在 3.10 版的變更: 新增*end_lineno* 與*end_offset* 屬性。

exception IndentationError

與不正確的縮排有關的語法錯誤的基礎類🄵。這是*SyntaxError* 的子類🄵。

exception TabError

當縮排包含🄵表符號 (tab) 和空白的不一致用法時會引發此例外。這是*IndentationError* 的子類🄵。

exception SystemError

當直譯器找到一個🄴部錯誤，但該情🄵看起來🄴有嚴重到要讓它放🄵所有的希望時會引發此例外。關聯值是一個表示什🄵地方出錯的字串（以低階的方式表達）。在*CPython* 中，這可能是因🄵錯誤地使用 Python 的 C API，例如回傳一個 `NULL` 值而🄵有設定例外。

如果你確定這個例外不是你的或者所用套件的錯，你應該向你的 Python 直譯器作者或維護者回報此錯誤。務必要回報該 Python 直譯器的版本（`sys.version`；這也會在互動式 Python 會話的開頭被印出）、確切的錯誤訊息（該例外的關聯值）及如果可能的話，觸發此錯誤的程式來源。

exception SystemExit

此例外會被*sys.exit()* 函式引發。它繼承自*BaseException* 而不是*Exception* 因此不會被捕捉*Exception* 的程式意外地捕捉。這允許例外可以正確地向上傳遞🄵導致直譯器結束。當它未被處理時，Python 直譯器會結束；不會印出堆🄵回溯。建構函式接受跟傳入*sys.exit()* 一樣的可選引數。如果該值是整數，它會指定系統的結束狀態（傳入 C 的 `exit()` 函式）；如果它是 `None`，結束狀態會是 0；如果它是其他型🄵（例如字串），則物件的值會被印出而結束狀態是 1。

對*sys.exit()* 的呼叫會轉譯成例外讓負責清理的處理函式（`try` 陳述式的 `finally` 子句）可以被執行，且讓除錯器可以在不冒著失去控制的風險下執行🄵本。如果在🄵對有必要立即結束的情🄵（例如在子行程呼叫完*os.fork()* 之後）可以使用*os._exit()* 函式。

code

傳入建構函式的結束狀態或錯誤訊息。（預設是 `None`。）

exception TypeError

當一個操作或函式被用在不適合的型🄵的物件時會引發此例外。關聯值是一個字串，提供關於不相符型🄵的細節。

此例外可能被使用者程式碼引發，以表示🄵不支援物件上所嘗試的操作，且本來就無意這樣做。如果一個物件有意要支援某個給定的操作但尚未提供實作，該引發的正確例外是*NotImplementedError*。

傳入錯誤型🄵的引數（例如當預期傳入*int* 🄵傳入*list*）應該要導致*TypeError*，但傳入帶有錯誤值的引數（例如超出預期範圍的數值）應該要導致*ValueError*。

exception UnboundLocalError

當在函式或方法🄵引用某個區域變數，但該變數尚未被🄵結到任何值的時候會引發此例外。這是*NameError* 的子類🄵。

exception UnicodeError

當 Unicode 相關的編碼或解碼錯誤發生時會引發此例外。這是*ValueError* 的子類🄴。

UnicodeError 有屬性描述編碼或解碼錯誤。例如 `err.object[err.start:err.end]` 會提供讓編解碼器失敗的具體無效輸入。

encoding

引發錯誤的編碼名稱。

reason

描述特定編解碼器錯誤的字串。

object

編解碼器嘗試編碼或解碼的物件。

start

在*object* 中無效資料的開始索引。

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

end

在*object* 中最後的無效資料後的索引。

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

exception UnicodeEncodeError

在編碼當中發生 Unicode 相關錯誤時會引發此例外。這是*UnicodeError* 的子類🄵。

exception UnicodeDecodeError

在解碼當中發生 Unicode 相關錯誤時會引發此例外。這是*UnicodeError* 的子類🄵。

exception UnicodeTranslateError

在轉譯當中發生 Unicode 相關錯誤時會引發此例外。這是*UnicodeError* 的子類🄵。

exception ValueError

當一個操作或函式收到引數是正確型🄵但是不適合的值，且該情🄵無法被更精確的例外例如*IndexError* 所描述時會引發此例外。

exception ZeroDivisionError

當除法或模數運算 (modulo operation) 的第二個引數是 0 的時候會引發此例外。關聯值是一個字串，表示運算元及運算的類型。

以下例外是🄵了相容於之前版本而保留；從 Python 3.3 開始，它們是*OSError* 的🄴名。

exception EnvironmentError

exception IOError

exception WindowsError

僅限於在 Windows 中使用。

5.4.1 作業系統例外

以下的例外是*OSError* 的子類🄵，它們根據系統錯誤代碼來引發。

exception BlockingIOError

當設置🄵非阻塞操作的物件（例如 socket）上的操作將要阻塞時會引發此例外。對應到 `errno` *EAGAIN*、*EALREADY*、*EWOULDBLOCK* 及*EINPROGRESS*。

除了*OSError* 的那些屬性之外，*BlockingIOError* 有多一個屬性：

characters_written

一個整數，🄵容🄵在其阻塞之前，已寫進串流的字元數。當使用*io* 模組🄵的緩衝 I/O 類🄵時這個屬性是可用的。

exception ChildProcessError

當子行程上的操作失敗時會引發此例外。對應到 `errno` *ECHILD*。

exception ConnectionError

連🄵相關問題的基礎類🄵。

子 類 🄵 有*BrokenPipeError*、*ConnectionAbortedError*、*ConnectionRefusedError* 及*ConnectionResetError*。

exception BrokenPipeError

ConnectionError 的子類🄵，當嘗試寫入管道 (pipe) 同時另一端已經被關閉時會引發此例外，或者當嘗試寫入已關閉寫入的 socket 時也會引發。對應到 `errno` *EPIPE* 及*ESHUTDOWN*。

exception ConnectionAbortedError

ConnectionError 的子類🄵。當一個連🄵的嘗試被對等端點 (peer) 中斷時會引發此例外。對應到 `errno` *ECONNABORTED*。

exception ConnectionRefusedError

ConnectionError 的子類🄵。當一個連🄵的嘗試被對等端點拒🄵時會引發此例外。對應到 `errno` *ECONNREFUSED*。

exception ConnectionResetError

ConnectionError 的子類🄵。當一個連🄵被對等端點重置時會引發此例外。對應到 errno *ECONNRESET*。

exception FileExistsError

當嘗試建立已存在的檔案或目🄵時會引發此例外。對應到 errno *EEXIST*。

exception FileNotFoundError

當請求不存在的檔案或目🄵時會引發此例外。對應到 errno *ENOENT*。

exception InterruptedError

當系統呼叫被傳入的信號中斷時會引發此例外。對應到 errno *EINTR*。

在 3.5 版的變更: 現在當 syscall 被信號中斷時 Python 會重試系統呼叫而不會引發*InterruptedError*，除非信號處理器引發例外（理由可參考 **PEP 475**）。

exception IsADirectoryError

當在目🄵上請求檔案操作（例如*os.remove()*）時會引發此例外。對應到 errno *EISDIR*。

exception NotADirectoryError

當在某個不是目🄵的東西上請求目🄵操作（例如*os.listdir()*）時會引發此例外。在大多數的 POSIX 平台上，如果嘗試操作開🄵或遍歷一個當作目🄵的非目🄵檔案也會引發此例外。對應到 errno *ENOTDIR*。

exception PermissionError

當嘗試執行一個🄵有合乎存取權限的操作時會引發此例外—例如檔案系統權限。對應到 errno *EACCES*、*EPERM* 及*ENOTCAPABLE*。

在 3.11.1 版的變更: WASI 的*ENOTCAPABLE* 現在對應到*PermissionError*。

exception ProcessLookupError

當給定的行程不存在時會引發此例外。對應到 errno *ESRCH*。

exception TimeoutError

當系統函式在系統層級超時會引發此例外。對應到 errno *ETIMEDOUT*。

在 3.3 版被加入: 加入以上所有的*OSError* 子類🄵。

也參考

PEP 3151 —改寫作業系統與 IO 例外階層

5.5 警告

以下的例外是當作警告的種類使用；更多細節參考警告類🄵 文件。

exception Warning

警告種類的基礎類🄵。

exception UserWarning

使用者程式碼🄵生的警告的基礎類🄵。

exception DeprecationWarning

關於已🄵用功能的警告的基礎類🄵，且當那些警告是針對其他 Python 開發者。

會被預設的警告過濾器忽略，在 __main__ 模組🄵除外 (**PEP 565**)。🄵用*Python 開發模式*會顯示此警告。

🄵用原則描述在 **PEP 387** 🄵。

exception PendingDeprecationWarning

關於過時且預期未來要被🄵用，但目前尚未被🄵用的功能的警告的基礎類🄵。

因🄵發出關於可能即將被🄵用的警告是不尋常的，此類🄵很少被使用，而對已經被🄵用的情🄵會優先使用*DeprecationWarning*。

會被預設的警告過濾器忽略。🄵用*Python 開發模式*會顯示此警告。

🄵用原則描述在 **PEP 387** 🄵。

exception SyntaxWarning

關於可疑語法的警告的基礎類🄵。

This warning is typically emitted when compiling Python source code, and usually won't be reported when running already compiled code.

exception RuntimeWarning

關於可疑執行環境行🄵的警告的基礎類🄵。

exception FutureWarning

關於已🄵用功能的警告的基礎類🄵，且當那些警告是針對以 Python 寫的應用程式的終端使用者。

exception ImportWarning

關於在模組引入的可能錯誤的警告的基礎類🄵。

會被預設的警告過濾器忽略。🄵用*Python 開發模式*會顯示此警告。

exception UnicodeWarning

Unicode 相關警告的基礎類🄵。

exception EncodingWarning

編碼相關警告的基礎類🄵。

細節參考*選擇性加入的編碼警告*。

在 3.10 版被加入.

exception BytesWarning

bytes 及*bytearray* 相關警告的基礎類🄵。

exception ResourceWarning

資源用法相關警告的基礎類🄵。

會被預設的警告過濾器忽略。🄵用*Python 開發模式*會顯示此警告。

在 3.2 版被加入.

5.6 例外群組

當需要引發多個不相關例外時會使用下列的類🄵。它們是例外階層的一部分所以可以像所有其他例外一樣使用 `except` 來處理。此外，它們會以包含的例外型🄵🄵基礎來比對其子群組而被 `except*` 辨認出來。

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

這兩個例外型🄵都將例外包裝在序列 `excs` 中。`msg` 參數必須是字串。這兩個類🄵的差🄵是*BaseExceptionGroup* 擴充了*BaseException* 且可以包裝任何例外，而*ExceptionGroup* 擴充了*Exception* 且只能包裝*Exception* 的子類🄵。這個設計使得 `except Exception` 可以捕捉*ExceptionGroup* 但不能捕捉*BaseExceptionGroup*。

如果所有包含的例外都是*Exception* 實例，*BaseExceptionGroup* 建構函式會回傳*ExceptionGroup* 而不是*BaseExceptionGroup*，因此可以被使用來讓這樣的選擇自動化。另一方面來🄵，如果任何包含的例外不是*Exception* 的子類🄵，*ExceptionGroup* 建構函式會引發*TypeError*。

message

建構函式的 msg 引數。這是一個唯讀的屬性。

exceptions

指定給建構函式 excs 序列中的例外組成的元組。這是一個唯讀的屬性。

subgroup(*condition*)

回傳只包含從現有群組比對到 *condition* 的例外的例外群組，或者當結果🄵空時回傳 None。

條件式可以是一個例外型🄵或是例外型🄵的元組，在此情🄵下，每個例外都會使用與 except 子句中使用的相同檢查方法來檢查是否有匹配。條件式也可以是一個可呼叫物件（除了型🄵物件之外），其接受一個例外作🄵單一引數，而如果該例外應該在子群組中就回傳 true。

現有例外的巢狀結構會保留在結果🄵，其*message*、*__traceback__*、*__cause__*、*__context__* 及*__notes__* 欄位的值也一樣。空的巢狀群組會從結果🄵排除。

條件會對巢狀例外群組🄵的所有例外做檢查，包括頂層及任何巢狀的例外群組。如果條件對這樣的例外群組🄵 true，它會被完整包含在結果🄵。

在 3.13 版被加入: condition 可以是任何不是型🄵物件的可呼叫物件。

split(*condition*)

像*subgroup()* 一樣，但回傳一對 (match, rest)，其中 match 是 subgroup(condition) 而 rest 是剩下🄵有比對到的部分。

derive(*excs*)

回傳有相同*message* 但將例外包裝在 excs 的例外群組。

此方法被*subgroup()* 及*split()* 使用，被用來在各種情境下拆分例外群組。子類🄵需要覆寫它來讓*subgroup()* 及*split()* 回傳子類🄵而不是*ExceptionGroup* 的實例。

subgroup() 及*split()* 會從原始的例外群組🄵🄵*__traceback__*、*__cause__*、*__context__* 和*__notes__* 欄位到*derive()* 所回傳的例外群組上，因此這些欄位不需要被*derive()* 更新。

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...     raise e
... except Exception as e:
...     exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'), Exception(
↪'cause'), ['a note'])
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'), ['a note
↪'])
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), ['a note
↪'])
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

需注意*BaseExceptionGroup* 定義了 __new__()，因此需要不同建構函式簽名的子類🄵需要覆寫它而不是 __init__()。例如下面定義了一個例外群組子類🄵接受 exit_code 🄵從中建構群組的訊息。:

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

像*ExceptionGroup*一樣，任何*BaseExceptionGroup*的子類🄴且也是*Exception*的子類🄴只能包裝*Exception*的實例。

在 3.11 版被加入.

5.7 例外階層

🄴建例外的類🄴階層如下：

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    ├── PythonFinalizationError
```

(繼續下一頁)

(繼續上一頁)

```
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

CHAPTER 6

文本處理 (Text Processing) 服務

本章節介紹的模組 (module) 提供了廣泛的字串操作與其他文本處理服務。

在二進位資料服務 下所描述的 *codecs* 模組也與文本處理高度相關。另外也請參🄵在*Text Sequence Type --- str* 中所描述的 Python 🄵建字串型🄵。

6.1 `string` --- 常見的字串操作

原始碼： Lib/string.py

也參考

Text Sequence Type --- str

字串方法

6.1.1 字串常數

此模組中定義的常數🄵：

`string.`**`ascii_letters`**

下文描述的 *ascii_lowercase* 和 *ascii_uppercase* 常數的串接，該值不依賴於區域設定。

`string.`**`ascii_lowercase`**

小寫字母 `'abcdefghijklmnopqrstuvwxyz'`。該值與地區設定無關且不會改變。

`string.`**`ascii_uppercase`**

大寫字母 `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`。此值與地區設定無關且不會改變。

`string.`**`digits`**

字串 `'0123456789'`。

`string.`**`hexdigits`**

字串 `'0123456789abcdefABCDEF'`。

string.**octdigits**

字串 '01234567'。

string.**punctuation**

在 C 語言中被視🄵標點符號的 ASCII 字元的字串：!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~。

string.**printable**

ASCII 字元的字串被 Python 是🄵可被列印輸出的。這是*digits*、*ascii_letters*、*punctuation* 和*whitespace* 的組合。

> **備🄵**
>
> By design, *string.printable.isprintable()* returns *False*. In particular, string.printable is not printable in the POSIX sense (see *LC_CTYPE*).

string.**whitespace**

包含所有 ASCII 字元的字串都視🄵空白字元 (whitespace)。包含空格 (space)、🄵表符號 (tab)、🄵行符號 (linefeed)、return、🄵頁符號 (formfeed) 和垂直🄵表符號 (vertical tab) 這些字元。

6.1.2 自訂字串格式

透過 **PEP 3101** 中描述的*format()* 方法，🄵建字串類🄵提供了進行🄵雜變數替🄵和數值格式化的能力。*string* 模組中的*Formatter* 類🄵模組可讓你使用與🄵建*format()* 方法相同的實作來建立和自訂你自己的字串格式化行🄵。

class string.**Formatter**

Formatter 類🄵有以下的公開方法：

format(*format_string*, */*, **args*, ***kwargs*)

主要的 API 方法。它接收一個格式字串及一組任意的位置引數與關鍵字引數，是呼叫*vformat()* 的包裝器 (wrapper)。

在 3.7 版的變更: 現在格式字串引數是僅限位置參數。

vformat(*format_string*, *args*, *kwargs*)

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the *args and **kwargs syntax. *vformat()* does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the *Formatter* defines a number of methods that are intended to be replaced by subclasses:

parse(*format_string*)

將 format_string 放入🄵圈，🄵回傳一個可🄵代物件，其元素🄵 (*literal_text*, *field_name*, *format_spec*, *conversion*)。這會被*vformat()* 用於將字串裁切🄵字面文本或替🄵欄位。

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be None. The value of *field_name* is unmodified and auto-numbering of non-numbered positional fields is done by *vformat()*.

get_field(*field_name*, *args*, *kwargs*)

Given *field_name*, convert it to an object to be formatted. Auto-numbering of *field_name* returned from *parse()* is done by *vformat()* before calling this method. Returns a tuple (obj, used_key). The default version takes strings of the form defined in **PEP 3101**, such as "0[name]" or "label.title". *args* and *kwargs* are as passed in to *vformat()*. The return value *used_key* has the same meaning as the *key* parameter to *get_value()*.

get_value(*key*, *args*, *kwargs*)

Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to *vformat()*, and the *kwargs* parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression '0.name' would cause *get_value()* to be called with a *key* argument of 0. The `name` attribute will be looked up after *get_value()* returns by calling the built-in *getattr()* function.

If the index or keyword refers to an item that does not exist, then an *IndexError* or *KeyError* should be raised.

check_unused_args(*used_args*, *args*, *kwargs*)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the *args* and *kwargs* that was passed to vformat. The set of unused args can be calculated from these parameters. *check_unused_args()* is assumed to raise an exception if the check fails.

format_field(*value*, *format_spec*)

format_field() simply calls the global *format()* built-in. The method is provided so that subclasses can override it.

convert_field(*value*, *conversion*)

Converts the value (returned by *get_field()*) given a conversion type (as in the tuple returned by the *parse()* method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.

6.1.3 格式化文字語法

The *str.format()* method and the *Formatter* class share the same syntax for format strings (although in the case of *Formatter*, subclasses can define their own format string syntax). The syntax is related to that of formatted string literals and template string literals, but it is less sophisticated and, in particular, does not support arbitrary expressions in interpolations.

Format strings contain "replacement fields" surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field: "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name:        arg_name ("." attribute_name | "[" element_index "]")*
arg_name:          [identifier | digit+]
attribute_name:    identifier
element_index:     digit+ | index_string
index_string:      <any source character except "]"> +
conversion:        "r" | "s" | "a"
format_spec:       format-spec:format_spec
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `'!'`, and a *format_spec*, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

另請參🄵格式規格 *(Format Specification)* 迷你語言 部份。

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. An *arg_name* is treated as a

number if a call to *str.isdecimal()* on the string would return true. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using *getattr()*, while an expression of the form `'[index]'` does an index lookup using `__getitem__()`.

在 3.1 版的變更: The positional argument specifiers can be omitted for *str.format()*, so `'{} {}'.format(a, b)` is equivalent to `'{0} {1}'.format(a, b)`.

在 3.4 版的變更: The positional argument specifiers can be omitted for *Formatter*.

Some simple format string examples:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the `__format__()` method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

目前支援三種轉🄵旗標: `'!s'` 會對該值呼叫*str()*, `'!r'` 會對該值呼叫*repr()*, 而 `'!a'` 則會對該值呼叫*ascii()*。

一些範例:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
"More {!a}"                      # Calls ascii() on the argument first
```

format_spec 欄位描述了值的呈現規格，例如欄位寬度、對齊、填充 (padding)、小數精度等細節資訊。每種值類型都可以定義自己的「格式化迷你語言 (formatting mini-language)」或對 *format_spec* 的解釋。

大多數🄵建型🄵都支援常見的格式化迷你語言，下一節將會詳細🄵明。

format_spec 欄位還可以在其🄵部包含巢狀的替🄵欄位。這些巢狀的替🄵欄位可能包含欄位名稱、轉🄵旗標、格式規格描述，但是不允許再更深層的巢狀結構。format_spec 🄵部的替🄵欄位會在 *format_spec* 字串被直譯前被替🄵。這讓數值的格式能🄵被動態地指定。

範例請見格式範例。

格式規格 (Format Specification) 迷你語言

「格式規格」在格式字串 (format string) 中包含的替🄵欄位中使用，以定義各個值如何被呈現（請參考格式化文字語法、f-strings 和 t-strings）。它們也能🄵直接傳遞給🄵建的*format()* 函式。每個可格式化型🄵 (formattable type) 可以定義格式規格如何被直譯。

大部分🄵建型🄵🄵了格式規格實作了下列選項，不過有些選項只被數值型🄵支援。

一般來🄵，輸入空格式規格會🄵生和對值呼叫*str()* 函式相同的結果，非空的格式規格才會修改結果。

標準格式🄵明符號 *(standard format specifier)* 的一般型式如下:

```
format_spec:             [options][width_and_precision][type]
options:                 [[fill]align][sign]["z"]["#"]["0"]
fill:                    <any character>
align:                   "<" | ">" | "=" | "^"
sign:                    "+" | "-" | " "
width_and_precision:     [width_with_grouping][precision_with_grouping]
width_with_grouping:     [width][grouping]
```

```
precision_with_grouping: "." [precision][grouping] | "." grouping
width:                   digit+
precision:               digit+
grouping:                "," | "_"
type:                    "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g"
                         | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

如果給定了一個有效的 *align* 值，則可以在它之前加一個 *fill* 字元，且該字元可F任意字元，若不加的話預設F空格。使用格式字串或*str.format()* 時是無法在其中使用大括號（"{" 或"}"）作F *fill* 字元的，但仍可透過巢狀替F欄位的方式插入大括號。此限制不影響*format()* 函式。

The meaning of the various alignment options is as follows:

選項	含義
'<'	F制欄位在可用空間F靠左對齊（這是大多數物件的預設值）。
'>'	F制欄位在可用空間F靠右對齊（這是數字的預設值）。
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types, excluding *complex*. It becomes the default for numbers when '0' immediately precedes the field width.
'^'	F制欄位在可用空間F置中。

請注意，除非有定義了最小欄位寬度，否則欄位寬度將始終與填充它的資料大小相同，故在該情F下的對齊選項是F有意義的。

sign 選項只適用於數字型F，F可F以下之一：

選項	含義
'+'	表示正數與負數均需使用符號。
'-'	表示正數應使用前導空格，負數應使用F號。
空格	表示正數應使用前導空格，負數應使用F號。

The 'z' option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

在 3.11 版的變更: 新增 'z' 選項（請見 **PEP 682**）。

The '#' option causes the "alternate form" to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the respective prefix '0b', '0o', '0x', or '0X' to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

width 是一個十進位整數，定義了最小總欄位寬度，包括任何前綴、分隔符號和其他格式字元。如果未指定，則欄位寬度將由F容F定。

當未給予明確的對齊指示，在 *width* 欄位前面填入零 ('0') 字元將會F*complex* 以外的數值型FF用有符號察覺的零填充 (sign-aware zero-padding)。這相當於使用 '0' F *fill* 字元且對齊類型F '='。

在 3.10 版的變更: 在 *width* 欄位前面加上 '0' 不再影響字串的預設對齊方式。

precision 是一個十進位整數，指定表示類型 'f' 和 'F' 的小數點後應顯示多少位，或表示類型 'g' 或 'G' 的小數點前後應顯示多少位。對於字串表示類型，該欄位指定最大欄位大小 - F言之，將使用欄位中的多少字元。整數表示類型不允許使用 *precision*。

The *grouping* option after *width* and *precision* fields specifies a digit group separator for the integral and fractional parts of a number respectively. It can be one of the following:

選項	含義
','	Inserts a comma every 3 digits for integer presentation type 'd' and floating-point presentation types, excluding 'n'. For other presentation types, this option is not supported.
'_'	在整數表示型🄵 'd' 和浮點數表示型🄵中，每 3 位數插入一個底🄵，這不包括 'n'。對於整數表示型🄵 'b'，'o'，'x' 和 'X'，每 4 位數字會插入底🄵。對於其他表示型🄵來🄵此選項🄵不支援。

For a locale aware separator, use the 'n' presentation type instead.

在 3.1 版的變更: 新增 ',' 選項（請見 **PEP 378**）。

在 3.6 版的變更: 新增 '_' 選項（請見 **PEP 515**）。

在 3.14 版的變更: Support the *grouping* option for the fractional part.

最終，型🄵🄵定了資料將會如何呈現

可用的字串表示型🄵有:

型🄵	含義
's'	String format. This is the default type for strings and may be omitted.
None	與 's' 相同。

The available integer presentation types are:

型🄵	含義
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	十六進位格式。輸出以 16 🄵基數的數字，9 以上的數字使用小寫字母。
'X'	十六進位格式。輸出以 16 🄵基數的數字，9 以上的數字使用大寫字母。如果指定了 '#'，則前綴 '0x' 也會被轉成大寫的 '0X'。
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate digit group separators.
None	與 'd' 相同。

In addition to the above presentation types, integers can be formatted with the floating-point presentation types listed below (except 'n' and None). When doing so, *float()* is used to convert the integer to a floating-point number before formatting.

The available presentation types for *float* and *Decimal* values are:

型 F	含義
`'e'`	Scientific notation. For a given precision `p`, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and `p` digits after the decimal point, for a total of `p + 1` significant digits. With no precision given, uses a precision of `6` digits after the decimal point for *`float`*, and shows all coefficient digits for *`Decimal`*. If `p=0`, the decimal point is omitted unless the `#` option is used.
`'E'`	Scientific notation. Same as `'e'` except it uses an upper case 'E' as the separator character.
`'f'`	Fixed-point notation. For a given precision `p`, formats the number as a decimal number with exactly `p` digits following the decimal point. With no precision given, uses a precision of `6` digits after the decimal point for *`float`*, and uses a precision large enough to show all coefficient digits for *`Decimal`*. If `p=0`, the decimal point is omitted unless the `#` option is used.
`'F'`	Fixed-point notation. Same as `'f'`, but converts `nan` to `NAN` and `inf` to `INF`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of `0` is treated as equivalent to a precision of `1`. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then, if `m <= exp < p`, where `m` is -4 for floats and -6 for *`Decimals`*, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the `'#'` option is used. With no precision given, uses a precision of `6` significant digits for *`float`*. For *`Decimal`*, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than `1e-6` in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision.
`'G'`	General format. Same as `'g'` except switches to `'E'` if the number gets too large. The representations of infinity and NaN are uppercased, too.
`'n'`	Number. This is the same as `'g'`, except that it uses the current locale setting to insert the appropriate digit group separators for the integral part of a number.
`'%'`	Percentage. Multiplies the number by 100 and displays in fixed (`'f'`) format, followed by a percent sign.
None	For *`float`* this is like the `'g'` type, except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point, and switches to the scientific notation when `exp >= p - 1`. When the precision is not specified, the latter will be as large as needed to represent the given value faithfully. For *`Decimal`*, this is the same as either `'g'` or `'G'` depending on the value of `context.capitals` for the current decimal context. The overall effect is to match the output of *`str()`* as altered by the other format modifiers.

The result should be correctly rounded to a given precision `p` of digits after the decimal point. The rounding mode for *`float`* matches that of the *`round()`* builtin. For *`Decimal`*, the rounding mode of the current *context* will be used.

The available presentation types for *`complex`* are the same as those for *`float`* (`'%'` is not allowed). Both the real and imaginary components of a complex number are formatted as floating-point numbers, according to the specified presentation type. They are separated by the mandatory sign of the imaginary part, the latter being terminated by a `j` suffix. If the presentation type is missing, the result will match the output of *`str()`* (complex numbers with a non-zero real part are also surrounded by parentheses), possibly altered by other format modifiers.

格式范例

本節包含*str.format()* 語法以及與舊式 `%` 格式的比較。

此語法在大多情🄵下與舊式的 `%` 格式類似，只是增加了 `{}` 和 `:` 來取代 `%`。例如，`'%03.2f'` 可以改寫🄵 `'{:03.2f}'`。

新的語法還支援新的選項，將在以下的範例中🄵明。

按位置存取引數：

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # 解包引數序列
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # 引數索引可以重🄵
'abracadabra'
```

按名稱存取引數：

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

存取引數的屬性：

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

存取引數的🄵容：

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替🄵 `%s` 和 `%r`：

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

對齊文字以及指定寬度：

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
```

(繼續下一頁)

(繼續上一頁)

```
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'**********centered**********'
```

替🄵 %+f、%-f 和 % f 以及指定正負號:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # 總會顯示
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # 若🄵正數則顯示空格
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # 只顯示負號 -- 與 '{:f}; {:f}' 相同
'3.140000; -3.140000'
```

替🄵 %x 和 %o 🄵將其值轉🄵🄵不同的進位制:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗號或底🄵作🄵數字分組分隔符號:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
>>> '{:_}'.format(1234567890)
'1_234_567_890'
>>> '{:_b}'.format(1234567890)
'100_1001_1001_0110_0000_0010_1101_0010'
>>> '{:_x}'.format(1234567890)
'4996_02d2'
>>> '{:_}'.format(123456789.123456789)
'123_456_789.12345679'
>>> '{:.,}'.format(123456789.123456789)
'123456789.123,456,79'
>>> '{:,._}'.format(123456789.123456789)
'123,456,789.123_456_79'
```

表示🄵百分比:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

作🄵特定型🄵格式:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

巢狀引數及更多🄵雜範例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
```

(繼續下一頁)

(繼續上一頁)

```
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 模板字串 ($-strings)

備🄵

The feature described here was introduced in Python 2.4; a simple templating method based upon regular expressions. It predates *`str.format()`*, formatted string literals, and *template string literals*.

It is unrelated to template string literals (t-strings), which were introduced in Python 3.14. These evaluate to *`string.templatelib.Template`* objects, found in the *`string.templatelib`* module.

模板字串提供如 **PEP 292** 所述更簡單的字串替🄵。模板字串的主要用例是國際化 (i18n)，因🄵在這種情🄵下，更簡單的語法和功能使得它比其他 Python 🄵建字串格式化工具更容易翻譯。基於模板字串建構的 i18n 函式庫範例，請參🄵 flufl.i18n 套件。

Template strings support `$`-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `$identifier` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` is restricted to any case-insensitive ASCII alphanumeric string (including underscores) that starts with an underscore or ASCII letter. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${identifier}` is equivalent to `$identifier`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `"${noun}ification"`.

Any other appearance of `$` in the string will result in a *`ValueError`* being raised.

The *`string`* module provides a *`Template`* class that implements these rules. The methods of *`Template`* are:

class string.**Template**(*template*)

The constructor takes a single argument which is the template string.

substitute(*mapping={}*, */*, ***kwds*)

進行模板替🄵，🄵回傳一個新的字串。*mapping* 是任何有金鑰🄵符合模板🄵🄵位符號的字典型物件。或者如果關鍵字就是🄵位符號時，你也可以改提供關鍵字引數。當 *mapping* 跟 *kwds* 同時給定🄵存在重🄵時，*kwds* 的🄵位符號會被優先使用。

safe_substitute(*mapping={}*, */*, ***kwds*)

類似於*`substitute()`*，但如果 *mapping* 與 *kwds* 中缺少🄵位符號的話，原始的🄵位符號會完整

地出現在結果字串🄵，而不會引發*KeyError* 例外。此外，與*substitute()* 不同的是，任何包含 `$` 的字句會直接回傳 `$` 而非引發*ValueError*。

雖然仍可能發生其他例外，但這個方法被認🄵是「安全」的，因🄵它總是試圖回傳一個有用的字串而不是引發例外。從另一個角度來看，*safe_substitute()* 可能🄵非完全安全，因🄵它會默默忽略格式錯誤的模板，這些模板包含了多余的左右定界符、不匹配的括號，或者不是有效的 Python 識🄵字的🄵位符號。

is_valid()

如果模板有將導致*substitute()* 引發*ValueError* 的無效🄵位符號，就會回傳 `False`。

在 3.11 版被加入.

get_identifiers()

回傳模板中有效識🄵字的串列，按照它們首次出現的順序，🄵忽略任何無效的識🄵字。

在 3.11 版被加入.

Template 實例也提供一個公開的資料屬性：

template

這是傳遞給建構函式 *template* 引數的物件。一般來🄵，你不應該改變它，但🄵🄵有🄵制設定成唯讀。

以下是如何使用 Template 的一個範例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

進階用法：你可以衍生*Template* 類🄵來自定義🄵位符號語法、左右定界符字元，或者用於剖析模板字串的正規表示式。你可以透過覆寫這些類🄵屬性來達成：

- *delimiter* -- 這是描述引入左右定界符的文字字串。預設值是 `$`。請注意這不是正規表示式，因🄵實作會在需要時對這個字串呼叫*re.escape()*。也請注意你不能在建立類🄵後修改左右定界符。（意即在子類🄵的命名空間中必須設置不同的左右定界符）
- *idpattern* -- This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression `(?a:[_a-z][_a-z0-9]*)`. If this is given and *braceidpattern* is `None` this pattern will also apply to braced placeholders.

  > **備🄵**
  >
  > Since default *flags* is `re.IGNORECASE`, pattern `[a-z]` can match with some non-ASCII characters. That's why we use the local `a` flag here.

  在 3.7 版的變更: *braceidpattern* can be used to define separate patterns used inside and outside the braces.

- *braceidpattern* -- This is like *idpattern* but describes the pattern for braced placeholders. Defaults to `None` which means to fall back to *idpattern* (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

在 3.7 版被加入.

- *flags* -- The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom *idpattern*s must follow conventions for verbose regular expressions.

  在 3.2 版被加入.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute *pattern*. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- *escaped* -- 此群組與跳🄵序列匹配，例如在預設模式下🄵 `$$`。
- *named* -- 此群組與不帶大括號的🄵位符號名稱匹配；它不應包含🄵取群組中的左右定界符號。
- *braced* -- 此群組與大括號括起來的🄵位符號名稱匹配；它不應在🄵取群組中包含左右定界符或大括號。
- *invalid* -- 此群組與任何其他左右定界符模式（通常是單一左右定界符）匹配，且它應該出現在正規表示式的最後。

當此模式有匹配於模板但這些命名組中🄵有任一個不匹配，此類🄵的方法將引發*`ValueError`*。

6.1.5 輔助函式

`string.`**`capwords`**(*s*, *sep=None*)

使用*`str.split()`* 將引數分割🄵字詞，使用*`str.capitalize()`* 將每個單字大寫，🄵使用*`str.join()`* 將大寫字詞連接起來。如果可選的第二引數 *sep* 不存在或🄵 `None`，則連續的空白字元將替🄵🄵單一空格，🄵且🄵除前導和尾隨空白；在其他情🄵下則使用 *sep* 來分割和連接單字。

6.2 `string.templatelib` --- 對模板字串字面值的支援

原始碼： Lib/string/templatelib.py

也參考

- Format strings
- Template string literal (t-string) syntax
- **PEP 750**

6.2.1 模板字串

在 3.14 版被加入.

Template strings are a mechanism for custom string processing. They have the full flexibility of Python's f-strings, but return a *`Template`* instance that gives access to the static and interpolated (in curly braces) parts of a string *before* they are combined.

To write a t-string, use a `'t'` prefix instead of an `'f'`, like so:

```
>>> pi = 3.14
>>> t't-strings are new in Python {pi!s}!'
Template(
   strings=('t-strings are new in Python ', '!'),
   interpolations=(Interpolation(3.14, 'pi', 's', ''),)
)
```

6.2.2 Types

class string.templatelib.**Template**

The `Template` class describes the contents of a template string. It is immutable, meaning that attributes of a template cannot be reassigned.

The most common way to create a `Template` instance is to use the template string literal syntax. This syntax is identical to that of f-strings, except that it uses a `t` prefix in place of an `f`:

```
>>> cheese = 'Red Leicester'
>>> template = t"We're fresh out of {cheese}, sir."
>>> type(template)
<class 'string.templatelib.Template'>
```

Templates are stored as sequences of literal *strings* and dynamic *interpolations*. A *values* attribute holds the values of the interpolations:

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
>>> template.interpolations
(Interpolation('Camembert', ...),)
>>> template.values
('Camembert',)
```

The `strings` tuple has one more element than `interpolations` and `values`; the interpolations "belong" between the strings. This may be easier to understand when tuples are aligned

```
template.strings:  ('Ah! We do have ',              '.')
template.values:   (                  'Camembert',    )
```

Attributes

strings: *tuple*[*str*, ...]

A *tuple* of the static strings in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
```

Empty strings *are* included in the tuple:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> template = t'Ah! {response}{cheese}.'
>>> template.strings
('Ah! ', '', '.')
```

The `strings` tuple is never empty, and always contains one more string than the `interpolations` and `values` tuples:

```
>>> t''.strings
('',)
>>> t''.values
()
>>> t'{'cheese'}'.strings
('', '')
>>> t'{'cheese'}'.values
('cheese',)
```

interpolations: *tuple*[*Interpolation*, ...]

A *tuple* of the interpolations in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.interpolations
(Interpolation('Camembert', 'cheese', None, ''),)
```

The `interpolations` tuple may be empty and always contains one fewer values than the `strings` tuple:

```
>>> t'Red Leicester'.interpolations
()
```

values: *tuple*[*object*, ...]

A tuple of all interpolated values in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.values
('Camembert',)
```

The `values` tuple always has the same length as the `interpolations` tuple. It is always equivalent to `tuple(i.value for i in template.interpolations)`.

方法

__new__(**args:* str | Interpolation)

While literal syntax is the most common way to create a `Template`, it is also possible to create them directly using the constructor:

```
>>> from string.templatelib import Interpolation, Template
>>> cheese = 'Camembert'
>>> template = Template(
...     'Ah! We do have ', Interpolation(cheese, 'cheese'), '.'
... )
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

If multiple strings are passed consecutively, they will be concatenated into a single value in the *strings* attribute. For example, the following code creates a *Template* with a single final string:

```
>>> from string.templatelib import Template
>>> template = Template('Ah! We do have ', 'Camembert', '.')
>>> template.strings
('Ah! We do have Camembert.',)
```

If multiple interpolations are passed consecutively, they will be treated as separate interpolations and an empty string will be inserted between them. For example, the following code creates a template with empty placeholders in the *strings* attribute:

```
>>> from string.templatelib import Interpolation, Template
>>> template = Template(
...     Interpolation('Camembert', 'cheese'),
...     Interpolation('.', 'punctuation'),
... )
>>> template.strings
('', '', '')
```

iter(template)

Iterate over the template, yielding each non-empty string and *Interpolation* in the correct order:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

警示

Empty strings are **not** included in the iteration:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> list(t'Ah! {response}{cheese}.')
['Ah! ',
 Interpolation('We do have ', 'response', None, ''),
 Interpolation('Camembert', 'cheese', None, ''),
 '.']
```

template + other

template += other

Concatenate this template with another, returning a new `Template` instance:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! ' + t'We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

Concatenating a `Template` and a `str` is **not** supported. This is because it is unclear whether the string should be treated as a static string or an interpolation. If you want to concatenate a `Template` with a string, you should either wrap the string directly in a `Template` (to treat it as a static string) or use an `Interpolation` (to treat it as dynamic):

```
>>> from string.templatelib import Interpolation, Template
>>> template = t'Ah! '
>>> # Treat 'We do have ' as a static string
>>> template += Template('We do have ')
>>> # Treat cheese as an interpolation
>>> cheese = 'Camembert'
>>> template += Template(Interpolation(cheese, 'cheese'))
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, '')]
```

class string.templatelib.**Interpolation**

The `Interpolation` type represents an expression inside a template string. It is immutable, meaning that attributes of an interpolation cannot be reassigned.

Interpolations support pattern matching, allowing you to match against their attributes with the match statement:

```
>>> from string.templatelib import Interpolation
>>> interpolation = t'{1. + 2.:.2f}'.interpolations[0]
>>> interpolation
Interpolation(3.0, '1. + 2.', None, '.2f')
>>> match interpolation:
...     case Interpolation(value, expression, conversion, format_spec):
...         print(value, expression, conversion, format_spec, sep=' | ')
...
3.0 | 1. + 2. | None | .2f
```

Attributes

value: *object*

The evaluated value of the interpolation.

```
>>> t'{1 + 2}'.interpolations[0].value
3
```

expression: *str*

The text of a valid Python expression, or an empty string.

The *expression* is the original text of the interpolation's Python expression, if the interpolation was created from a t-string literal. Developers creating interpolations manually should either set this to an empty string or choose a suitable valid Python expression.

```
>>> t'{1 + 2}'.interpolations[0].expression
'1 + 2'
```

conversion: *Literal*['a', 'r', 's'] | *None*

The conversion to apply to the value, or `None`.

The `conversion` is the optional conversion to apply to the value:

```
>>> t'{1 + 2!a}'.interpolations[0].conversion
'a'
```

> **備🄵**
>
> Unlike f-strings, where conversions are applied automatically, the expected behavior with t-strings is that code that *processes* the `Template` will decide how to interpret and whether to apply the `conversion`. For convenience, the *convert()* function can be used to mimic f-string conversion semantics.

format_spec: *str*

The format specification to apply to the value.

The `format_spec` is an optional, arbitrary string used as the format specification to present the value:

```
>>> t'{1 + 2:.2f}'.interpolations[0].format_spec
'.2f'
```

> **備🄵**
>
> Unlike f-strings, where format specifications are applied automatically via the *format()* protocol, the expected behavior with t-strings is that code that *processes* the interpolation will decide how to interpret and whether to apply the format specification. As a result, `format_spec` values in interpolations can be arbitrary strings, including those that do not conform to the *format()* protocol.

方法

__new__(*value:* object, *expression:* str, *conversion:* Literal*['a', 'r', 's']* | None = *None*, *format_spec:* str = '')

Create a new `Interpolation` object from component parts.

參數

- **value** -- The evaluated, in-scope result of the interpolation.
- **expression** -- The text of a valid Python expression, or an empty string.
- **conversion** -- The *conversion* to be used, one of `None`, `'a'`, `'r'`, or `'s'`.

- **format_spec** -- An optional, arbitrary string used as the *format specification* to present the value.

6.2.3 Helper functions

string.templatelib.**convert**(*obj*, */*, *conversion*)

Applies formatted string literal *conversion* semantics to the given object *obj*. This is frequently useful for custom template string processing logic.

Three conversion flags are currently supported:

- `'s'` which calls *str()* on the value (like `!s`),
- `'r'` which calls *repr()* (like `!r`), and
- `'a'` which calls *ascii()* (like `!a`).

If the conversion flag is `None`, *obj* is returned unchanged.

6.3 re --- 正規表示式 (regular expression) 操作

原始碼： Lib/re/

此模組提供類似於 Perl 中正規表示式的配對操作。

被搜尋的模式 (pattern) 與字串可以是 Unicode 字串 (*str*)，也可以是 8-bit 字串 (*bytes*)。然而，Unicode 字串和 8-bit 字串不能混用：也就是，你不能用 byte 模式配對 Unicode 字串，反之亦然；同樣地，替🄵時，用來替🄵的字串必須與模式和搜尋字串是相同的型🄵 (type)。

正規表示式使用反斜🄵字元 (`'\'`) 表示特🄵的形式，或是使用特殊字元而不呼叫它們的特殊意義。這與 Python 在字串文本 (literal) 中，🄵了一樣的目的使用同一個字元的目的相衝突；舉例來🄵，🄵了配對一個反斜🄵文字，一個人可能需要寫 `'\\\\'` 當作模式字串，因🄵正規表示式必須是 `\\`，而且每個反斜🄵在一個普通的 Python 字串文本中必須表示🄵 `\\`。另外，請注意在 Python 的字串文本中使用反斜🄵的任何無效跳🄵序列目前會🄵生一個*SyntaxWarning*，而在未來這會變成一個*SyntaxError*。🄵管它對正規表示式是一個有效的跳🄵序列，這種行🄵也會發生。

解🄵方法是對正規表示式模式使用 Python 的原始字串符號；反斜🄵在一個以 `'r'` 🄵前綴的字串文本中不會被用任何特🄵的方式處理。所以 `r"\n"` 是一個兩個字元的字串，包含 `'\'` 和 `'n'`，同時 `"\n"` 是一個單個字元的字串，包含一個🄵行符號。通常模式在 Python 程式中會使用這個原始字串符號表示。

請務必注意到大部分的正規表示式操作是可以在模組層級的函式和*compiled regular expressions* 中的方法使用的。這些函式是個捷徑且讓你不需要先編譯一個正規表示式物件，但是會缺少一些微調參數。

也參考

第三方的 regex 模組，有著和標準函式庫*re* 模組相容的 API，但是提供額外的功能以及更完整的 Unicode 支援。

6.3.1 正規表示式語法

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. In general, if a string *p* matches *A* and another string *q* matches *B*, the string *pq* will match AB. This holds unless *A* or *B* contain low precedence operations; boundary conditions between *A* and *B*; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [Frie09], or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the regex-howto.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string `'last'`. (In the rest of this section, we'll write RE's in `this special style`, usually without quotes, and strings to be matched `'in single quotes'`.)

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Repetition operators or quantifiers (`*`, `+`, `?`, `{m,n}`, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression `(?:a{6})*` matches any multiple of six `'a'` characters.

The special characters are:

.

(Dot.) In the default mode, this matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline. `(?s:.)` matches any character regardless of flags.

^

(Caret.) Matches the start of the string, and in *MULTILINE* mode also matches immediately after each newline.

$

Matches the end of the string or just before the newline at the end of the string, and in *MULTILINE* mode also matches before a newline. `foo` matches both 'foo' and 'foobar', while the regular expression `foo$` matches only 'foo'. More interestingly, searching for `foo.$` in `'foo1\nfoo2\n'` matches 'foo2' normally, but 'foo1' in *MULTILINE* mode; searching for a single `$` in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

*****

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. `ab*` will match 'a', 'ab', or 'a' followed by any number of 'b's.

+

Causes the resulting RE to match 1 or more repetitions of the preceding RE. `ab+` will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

?

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either 'a' or 'ab'.

***?, +?, ??**

The `'*'`, `'+'`, and `'?'` quantifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE `<.*>` is matched against `'<a> b <c>'`, it will match the entire string, and not just `'<a>'`. Adding `?` after the quantifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE `<.*?>` will match only `'<a>'`.

***+, ++, ?+**

Like the `'*'`, `'+'`, and `'?'` quantifiers, those where `'+'` is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, `a*a` will match `'aaaa'` because the `a*` will match all 4 `'a'`s, but, when the final `'a'` is encountered, the expression is backtracked so that in the end the `a*` ends up matching 3 `'a'`s total, and the fourth `'a'` is matched by the final `'a'`. However, when `a*+a` is used to match `'aaaa'`, the `a*+` will match all 4 `'a'`, but when the final `'a'` fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. `x*+`, `x++` and `x?+` are equivalent to `(?>x*)`, `(?>x+)` and `(?>x?)` correspondingly.

在 3.11 版被加入.

{m}

Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

{m,n}
Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, `a{3,5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, `a{4,}b` will match `'aaaab'` or a thousand `'a'` characters followed by a `'b'`, but not `'aaab'`. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?
Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous quantifier. For example, on the 6-character string `'aaaaaa'`, `a{3,5}` will match 5 `'a'` characters, while `a{3,5}?` will only match 3 characters.

{m,n}+
Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string `'aaaaaa'`, `a{3,5}+aa` attempt to match 5 `'a'` characters, then, requiring 2 more `'a'`s, will need more characters than available and thus fail, while `a{3,5}aa` will match with `a{3,5}` capturing 5, then 4 `'a'`s by backtracking and then the final 2 `'a'`s are matched by the final `aa` in the pattern. `x{m,n}+` is equivalent to `(?>x{m,n})`.

在 3.11 版被加入.

****
Either escapes special characters (permitting you to match characters like `'*'`, `'?'`, and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[]
Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `'a'`, `'m'`, or `'k'`.
- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it's placed as the first or last character (e.g. `[-a]` or `[a-]`), it will match a literal `'-'`.
- Special characters except backslash lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.
- Backslash either escapes characters which have special meaning in a set such as `'-'`, `']'`, `'^'` and `'\\'` itself or signals a special sequence which represents a single character such as `\xa0` or `\n` or a character class such as `\w` or `\S` (defined below). Note that `\b` represents a single "backspace" character, not a word boundary as outside a set, and numeric escapes such as `\1` are always octal escapes, not group references. Special sequences which do not match a single character such as `\A` and `\z` are not allowed.
- Characters that are not within a range can be matched by *complementing* the set. If the first character of the set is `'^'`, all the characters that are *not* in the set will be matched. For example, `[^5]` will match any character except `'5'`, and `[^^]` will match any character except `'^'`. `^` has no special meaning if it's not the first character in the set.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- Support of nested sets and set operations as in Unicode Technical Standard #18 might be added in the future. This would change the syntax, so to facilitate this change a *FutureWarning* will be raised in ambiguous cases for the time being. That includes sets starting with a literal `'['` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning escape them with a backslash.

在 3.7 版的變更: *FutureWarning* is raised if a character set contains constructs that will change semantically in the future.

|

`A|B`, where *A* and *B* can be arbitrary REs, creates a regular expression that will match either *A* or *B*. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once *A* matches, *B* will not be tested further, even if it would produce a longer overall match. In other words, the `'|'` operator is never greedy. To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

(...)

Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals `'('` or `')'`, use `\(` or `\)`, or enclose them inside a character class: `[(]`, `[)]`.

(?...)

This is an extension notation (a `'?'` following a `'('` is not meaningful otherwise). The first character after the `'?'` determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; `(?P<name>...)` is the only exception to this rule. Following are the currently supported extensions.

(?aiLmsux)

(One or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) The group matches the empty string; the letters set the corresponding flags for the entire regular expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in 模組🄵容.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the *re.compile()* function. Flags should be used first in the expression string.

在 3.11 版的變更: This construction can only be used at the start of the expression.

(?:...)

A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group *cannot* be retrieved after performing a match or referenced later in the pattern.

(?aiLmsux-imsx:...)

(Zero or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`, optionally followed by `'-'` followed by one or more letters from the `'i'`, `'m'`, `'s'`, `'x'`.) The letters set or remove the corresponding flags for the part of the expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in 模組𝔽容.)

The letters `'a'`, `'L'` and `'u'` are mutually exclusive when used as inline flags, so they can't be combined or follow `'-'`. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns `(?a:...)` switches to ASCII-only matching, and `(?u:...)` switches to Unicode matching (default). In bytes patterns `(?L:...)` switches to locale dependent matching, and `(?a:...)` switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

在 3.6 版被加入.

在 3.7 版的變更: The letters `'a'`, `'L'` and `'u'` also can be used in a group.

(?>...)
Attempts to match `...` as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the `(?>...)` because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, `(?>.*).` would never match anything because first the `.*` would match all characters possible, then, having nothing left to match, the final `.` would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

在 3.11 版被加入.

(?P<name>...)
Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in *bytes* patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is `(?P<quote>['"]).*?(?P=quote)` (i.e. matching a string quoted with either single or double quotes):

Context of reference to group "quote"	Ways to reference it
in the same pattern itself	• `(?P=quote)` (as shown) • `\1`
when processing match object *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
in a string passed to the *repl* argument of `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

在 3.12 版的變更: In *bytes* patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

(?P=name)
A backreference to a named group; it matches whatever text was matched by the earlier group named *name*.

(?#...)
A comment; the contents of the parentheses are simply ignored.

(?=...)
Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

(?!...)
Matches if `...` doesn't match next. This is a *negative lookahead assertion*. For example, `Isaac (?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

(?<=...)
Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(?<=abc)def` will find a match in `'abcdef'`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the *search()* function rather than the *match()* function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

在 3.5 版的變更: Added support for group references of fixed length.

(?<!...)
Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern)
Will try to match with `yes-pattern` if the group with given *id* or *name* exists, and with `no-pattern` if it doesn't. `no-pattern` is optional and can be omitted. For example, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` is a poor email matching pattern, which will match with `'<user@host.com>'` as well as `'user@host.com'`, but not with `'<user@host.com'` nor `'user@host.com>'`.

在 3.12 版的變更: Group *id* can only contain ASCII digits. In *bytes* patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

\number
Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

\A
Matches only at the start of the string.

\b
Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning or end of the string. This means that `r'\bat\b'` matches `'at'`, `'at.'`, `'(at)'`, and `'as at ay'` but not `'attempt'` or `'atlas'`.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the *ASCII* flag. Word boundaries are determined by the current locale if the *LOCALE* flag is used.

> **備🄵**

Inside a character range, `\b` represents the backspace character, for compatibility with Python's string literals.

\B

Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'at\B'` matches `'athens'`, `'atom'`, `'attorney'`, but not `'at'`, `'at.'`, or `'at!'`. `\B` is the opposite of `\b`, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the *ASCII* flag. Word boundaries are determined by the current locale if the *LOCALE* flag is used.

在 3.14 版的變更: `\B` now matches empty input string.

\d

For Unicode (str) patterns:
Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters.

Matches `[0-9]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches any decimal digit in the ASCII character set; this is equivalent to `[0-9]`.

\D

Matches any character which is not a decimal digit. This is the opposite of `\d`.

Matches `[^0-9]` if the *ASCII* flag is used.

\s

For Unicode (str) patterns:
Matches Unicode whitespace characters (as defined by *str.isspace()*). This includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

Matches `[ \t\n\r\f\v]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches characters considered whitespace in the ASCII character set; this is equivalent to `[ \t\n\r\f\v]`.

\S

Matches any character which is not a whitespace character. This is the opposite of `\s`.

Matches `[^ \t\n\r\f\v]` if the *ASCII* flag is used.

\w

For Unicode (str) patterns:
Matches Unicode word characters; this includes all Unicode alphanumeric characters (as defined by *str.isalnum()*), as well as the underscore (_).

Matches `[a-zA-Z0-9_]` if the *ASCII* flag is used.

For 8-bit (bytes) patterns:
Matches characters considered alphanumeric in the ASCII character set; this is equivalent to `[a-zA-Z0-9_]`. If the *LOCALE* flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W

Matches any character which is not a word character. This is the opposite of `\w`. By default, matches non-underscore (_) characters for which *str.isalnum()* returns `False`.

Matches `[^a-zA-Z0-9_]` if the *ASCII* flag is used.

If the *LOCALE* flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\z
Matches only at the end of the string.

在 3.14 版被加入.

\Z
The same as `\z`. For compatibility with old Python versions.

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(Note that `\b` is used to represent word boundaries, and means "backspace" only inside character classes.)

`'\u'`, `'\U'`, and `'\N'` escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

在 3.3 版的變更: The `'\u'` and `'\U'` escape sequences have been added.

在 3.6 版的變更: Unknown escapes consisting of `'\'` and an ASCII letter now are errors.

在 3.8 版的變更: The `'\N{name}'` escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. `'\N{EM DASH}'`).

6.3.2 模組🄵容

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

旗標

在 3.6 版的變更: Flag constants are now instances of *RegexFlag*, which is a subclass of *enum.IntFlag*.

class re.**RegexFlag**
An *enum.IntFlag* class containing the regex options listed below.

在 3.11 版被加入: - added to `__all__`

re.**A**

re.**ASCII**
Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag `(?a)`.

> **備🄵**
>
> The *U* flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for `str` patterns, and Unicode matching isn't allowed for bytes patterns. *UNICODE* and the inline flag `(?u)` are similarly redundant.

re.**DEBUG**
Display debug information about compiled expression.

No corresponding inline flag.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like `[A-Z]` will also match lowercase letters. Full Unicode matching (such as `Ü` matching `ü`) also works unless the *ASCII* flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the *LOCALE* flag is also used.

Corresponds to the inline flag `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the *IGNORECASE* flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). If the *ASCII* flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched.

re.**L**

re.**LOCALE**

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag `(?L)`.

> **警告**
>
> This flag is discouraged; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one "culture" at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

在 3.6 版的變更: *LOCALE* can be used only with bytes patterns and is not compatible with *ASCII*.

在 3.7 版的變更: Compiled regular expression objects with the *LOCALE* flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.**M**

re.**MULTILINE**

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag `(?m)`.

re.**NOFLAG**

Indicates no flag being applied, the value is `0`. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

在 3.11 版被加入.

re.**S**

re.**DOTALL**

Make the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

Corresponds to the inline flag `(?s)`.

re.**U**

re.**UNICODE**

In Python 3, Unicode characters are matched by default for `str` patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See *ASCII* to restrict matching to ASCII characters instead.

re.**X**

re.**VERBOSE**

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?`, `(?:` or `(?P<...>`. For example, `(? :` and `* ?` are not allowed. When a line contains a `#` that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such `#` through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponds to the inline flag `(?x)`.

函式

re.**compile**(*pattern*, *flags=0*)

將正規表示式模式編譯成正規表示式物件，可以使用它的*match()*、*search()* 等方法來匹配，如下所述。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、`|` 運算子）組合。

順序🄵：

```
prog = re.compile(pattern)
result = prog.match(string)
```

等價於：

```
result = re.match(pattern, string)
```

但是當表示式在單一程式中多次使用時，使用*re.compile()* 🄵保存🄵生的正規表示式物件以供重🄵使用會更有效率。

> **備🄵**
>
> 傳遞給*re.compile()* 之最新模式的編譯版本和模組層級匹配函式都會被快取，因此一次僅使用幾個正規表示式的程式不必去擔心編譯正規表示式。

re.**search**(*pattern*, *string*, *flags=0*)

掃描 *string* 以尋找正規表示式 *pattern* 🄵生匹配的第一個位置，🄵回傳對應的*Match*。如果字串中🄵有與模式匹配的位置則回傳 `None`；請注意，這與在字串中的某個點查找零長度匹配不同。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、`|` 運算子）組合。

re.**match**(*pattern*, *string*, *flags=0*)

如果 *string* 開頭的零個或多個字元與正規表示式 *pattern* 匹配，則回傳對應的*Match*。如果字串與模式不匹配，則回傳 `None`；請注意，這與零長度匹配不同。

請注意，即使在*MULTILINE* 模式 (mode) 下，*re.match()* 只會於字串的開頭匹配，而非每行的開頭。

如果你想在 *string* 中的任何位置找到匹配項，請使用*search()*（另請參🄵*search() vs. match()*）。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

`re.`**`fullmatch`**(*pattern*, *string*, *flags=0*)

如果整個 *string* 與正規表示式 *pattern* 匹配，則回傳對應的`Match`。如果字串與模式不匹配，則回傳 `None`；請注意，這與零長度匹配不同。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

在 3.4 版被加入.

`re.`**`split`**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

依 *pattern* 的出現次數拆分 *string*。如果在 *pattern* 中使用捕獲括號，則模式中所有群組的文字也會作🄵結果串列的一部分回傳。如果 *maxsplit* 非零，則最多發生 *maxsplit* 次拆分，🄵且字串的其余部分會作🄵串列的最終元素回傳。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

如果分隔符號中有捕獲群組🄵且它在字串的開頭匹配，則結果將以空字串開頭。這同樣適用於字串的結尾：

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

如此一來，分隔符號元件始終可以在結果串列中的相同相對索引處找到。

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

在 3.1 版的變更: 新增可選的旗標引數。

在 3.7 版的變更: 新增了對可以匹配空字串之模式進行拆分的支援。

在 3.13 版之後被🄵用: 將 *maxsplit* 和 *flags* 作🄵位置引數傳遞的用法已被🄵用。在未來的 Python 版本中，它們將會是僅限關鍵字參數。

`re.`**`findall`**(*pattern*, *string*, *flags=0*)

以字串或元組串列的形式回傳 *string* 中 *pattern* 的所有非重🄵匹配項。從左到右掃描 *string*，🄵按找到的順序回傳符合項目。結果中會包含空匹配項。

結果取🄵於模式中捕獲群組的數量。如果🄵有群組，則回傳與整個模式匹配的字串串列。如果恰好存在一個群組，則回傳與該群組匹配的字串串列。如果存在多個群組，則回傳與群組匹配的字串元組串列。非捕獲群組則不影響結果的形式。

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

在 3.7 版的變更: 非空匹配現在可以剛好在前一個空匹配的後面開始。

`re.`**`finditer`**(*pattern*, *string*, *flags=0*)

回傳一個*iterator*，在 *string* 中的 RE *pattern* 的所有非重🄵匹配上 yield `Match` 物件。從左到右掃描 *string*，🄵按找到的順序回傳匹配項目。結果中包含空匹配項。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

在 3.7 版的變更: 非空匹配現在可以剛好在前一個空匹配的後面開始。

`re.`**`sub`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

回傳透過以替🄵 *repl* 取代 *string* 中最左邊不重🄵出現的 *pattern* 所獲得的字串。如果未找到該模式，則不改變 *string* 🄵回傳。*repl* 可以是字串或函式；如果它是字串，則處理其中的任何反斜🄵跳🄵。也就是🄵 `\n` 會被轉🄵🄵單一🄵行符、`\r` 會被轉🄵🄵回車符 (carriage return) 等等。ASCII 字母的未知跳🄵符會被保留以供將來使用🄵被視🄵錯誤。其他未知跳🄵符例如 `\&` 會被單獨保留。例如 `\6` 的反向參照將被替🄵🄵模式中第 6 組匹配的子字串。例如：

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl* 是一個函式，則 *pattern* 的每個不重🄵出現之處都會呼叫它。此函式接收單一`Match` 引數，🄵回傳替🄵字串。例如：

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

此模式可以是字串或`Pattern`。

可選引數 *count* 是要替🄵的模式出現的最大次數；*count* 必須是非負整數。如果省略或🄵零，則所有出現的🄵容都將被替🄵。

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match. As a result, `sub('x*', '-', 'abxd')` returns `'-a-b--d-'` instead of `'-a-b-d-'`.

在字串型🄵 *repl* 引數中，除了上述字元跳🄵和反向參照之外，`\g<name>` 將使用名🄵 `name` 的群組所匹配到的子字串，如 `(?P<name>...)` 所定義的語法。`\g<number>` 使用對應的群組編號；因此 `\g<2>` 等價於 `\2`，但在諸如 `\g<2>0` 之類的替🄵中🄵非模糊不清 (isn't ambiguous)。`\20` 將被直譯🄵對群組 20 的參照，而不是對後面跟著字面字元 `'0'` 的群組 2 的參照。反向參照 `\g<0>` 會取代以 RE 所匹配到的整個子字串。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

在 3.1 版的變更: 新增可選的旗標引數。

在 3.5 版的變更: 不匹配的群組將被替🄵🄵空字串。

在 3.6 版的變更: 在由 `'\'` 和一個 ASCII 字母組成之 *pattern* 中的未知跳🄵符現在🄵錯誤。

在 3.7 版的變更: 由 `'\'` 和一個 ASCII 字母組成之 *repl* 中的未知跳🄵符現在🄵錯誤。非空匹配後可以緊接著出現空匹配。

在 3.12 版的變更: 群組 *id* 只能包含 ASCII 數字。在`bytes` 替🄵字串中，群組 *name* 只能包含 ASCII 範圍🄵的位元組 (`b'\x00'-b'\x7f'`)。

在 3.13 版之後被🄵用: 將 *count* 和 *flags* 作🄵位置引數傳遞的用法已被🄵用。在未來的 Python 版本中，它們將會是僅限關鍵字參數。

re.**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

執行與*sub()* 相同的操作，但回傳一個元組 (new_string, number_of_subs_made)。

可以透過指定 *flags* 值來修改運算式的行🄵，值可以是任何*flags* 變數，使用位元 OR（bitwise OR、| 運算子）組合。

re.**escape**(*pattern*)

對 *pattern* 中的特殊字元進行跳🄵。如果你想要匹配其中可能包含正規表示式元字元 (metacharacter) 的任意文本字串，這會非常有用。例如:

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

此函式不得用於*sub()* 和*subn()* 中的替🄵字串，僅應跳🄵反斜🄵。例如:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

在 3.3 版的變更: '_' 字元不再被跳🄵。

在 3.7 版的變更: Only characters that can have special meaning in a regular expression are escaped. As a result, '!', '"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@', and "`" are no longer escaped.

re.**purge**()

清除正規表示式快取。

例外

exception re.**PatternError**(*msg*, *pattern=None*, *pos=None*)

當傳遞給此處函式之一的字串不是有效的正規表示式（例如它可能包含不匹配的括號）或在編譯或匹配期間發生某些其他錯誤時，將引發例外。如果字串不包含模式匹配項，則🄵不是錯誤。PatternError 實例具有以下附加屬性:

msg

未格式化的錯誤訊息。

pattern

正規表示式模式。

pos

pattern 中編譯失敗的索引（可能是 None）。

lineno

對應 *pos* 的列（可能是 None）。

colno

對應 *pos* 的欄（可能是 None）。

在 3.5 版的變更: 新增額外屬性。

在 3.13 版的變更: PatternError 最初被命名🄵 error；後者🄵了向後相容性而被保留🄵🄵名。

6.3.3 Regular Expression Objects

class re.**Pattern**

Compiled regular expression object returned by *re.compile()*.

在 3.9 版的變更: *re.Pattern* supports `[]` to indicate a Unicode (str) or bytes pattern. See 泛型🄵名型🄵.

Pattern.**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding *Match*. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match. If *endpos* is less than *pos*, no match will be found; otherwise, if *rx* is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding *Match*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use *search()* instead (see also *search() vs. match()*).

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding *Match*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

在 3.4 版被加入.

Pattern.**split**(*string*, *maxsplit=0*)

Identical to the *split()* function, using the compiled pattern.

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

Similar to the *findall()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

Similar to the *finditer()* function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for *search()*.

Pattern.**sub**(*repl*, *string*, *count=0*)

Identical to the *sub()* function, using the compiled pattern.

Pattern.**subn**(*repl*, *string*, *count=0*)

Identical to the *subn()* function, using the compiled pattern.

Pattern.**flags**

The regex matching flags. This is a combination of the flags given to *compile()*, any `(?...)` inline flags in the pattern, and implicit flags such as *UNICODE* if the pattern is a Unicode string.

Pattern.**groups**

The number of capturing groups in the pattern.

Pattern.**groupindex**

A dictionary mapping any symbolic group names defined by `(?P<id>)` to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

Pattern.**pattern**

The pattern string from which the pattern object was compiled.

在 3.7 版的變更: Added support of *copy.copy()* and *copy.deepcopy()*. Compiled regular expression objects are considered atomic.

6.3.4 Match Objects

Match objects always have a boolean value of `True`. Since *match()* and *search()* return `None` when there is no match, you can test whether there was a match with a simple `if` statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

class re.**Match**

Match object returned by successful `match`es and `search`es.

在 3.9 版的變更: *re.Match* supports `[]` to indicate a Unicode (str) or bytes match. See 泛型🄵名型🄵.

Match.**expand**(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the *sub()* method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences (`\1`, `\2`) and named backreferences (`\g<1>`, `\g<name>`) are replaced by the contents of the corresponding group. The backreference `\g<0>` will be replaced by the entire match.

在 3.5 版的變更: 不匹配的群組將被替🄵🄵空字串。

Match.**group**([*group1*, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an *IndexError* exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
```

(繼續下一頁)

(繼續上一頁)

```
'Newton'
>>> m.group(1, 2)     # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an *IndexError* exception is raised.

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

`Match.__getitem__`(*g*)

This is identical to `m.group(g)`. This allows easier access to an individual group from a match:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Named groups are supported as well:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
>>> m['first_name']
'Isaac'
>>> m['last_name']
'Newton'
```

在 3.6 版被加入.

`Match.groups`(*default=None*)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to `None`.

舉例來🄵:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.**groupdict**(*default=None*)

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])

Match.**end**([*group*])

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return `-1` if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to `m.group(g)`) is

```
m.string[m.start(g):m.end(g)]
```

Note that `m.start(group)` will equal `m.end(group)` if *group* matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an *IndexError* exception.

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])

For a match *m*, return the 2-tuple `(m.start(group), m.end(group))`. Note that if *group* did not contribute to the match, this is `(-1, -1)`. *group* defaults to zero, the entire match.

Match.**pos**

The value of *pos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string at which the RE engine started looking for a match.

Match.**endpos**

The value of *endpos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string beyond which the RE engine will not go.

Match.**lastindex**

The integer index of the last matched capturing group, or `None` if no group was matched at all. For example, the expressions `(a)b`, `((a)(b))`, and `((ab))` will have `lastindex == 1` if applied to the string `'ab'`, while the expression `(a)(b)` will have `lastindex == 2`, if applied to the same string.

Match.**lastgroup**

The name of the last matched capturing group, or `None` if the group didn't have a name, or if no group was matched at all.

Match.**re**

The *regular expression object* whose *match()* or *search()* method produced this match instance.

Match.**string**

The string passed to *match()* or *search()*.

在 3.7 版的變更: Added support of *copy.copy()* and *copy.deepcopy()*. Match objects are considered atomic.

6.3.5 Regular Expression Examples

Checking for a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, "a" for ace, "k" for king, "q" for queen, "j" for jack, "t" for 10, and "2" through "9" representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, `"727ak"`, contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the *group()* method of the match object in the following manner:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

模擬 scanf()

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

scanf() Token	Regular Expression
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x、%X	[-+]?(0[xX])?[\dA-Fa-f]+

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python offers different primitive operations based on regular expressions:

- *re.match()* checks for a match only at the beginning of the string
- *re.search()* checks for a match anywhere in the string (this is what Perl does by default)
- *re.fullmatch()* checks for entire string to be a match

舉例來🄵:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

Regular expressions beginning with `'^'` can be used with *search()* to restrict the match at the beginning of the string:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Note however that in *MULTILINE* mode *match()* only matches at the beginning of the string, whereas using *search()* with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Making a Phonebook

split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of *split()* because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The `:?` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of `4`, we could separate the house number from the street name:

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Text Munging

sub() replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using *sub()* with a function to ”munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Finding all Adverbs

findall() matches *all* occurrences of a pattern, not just the first one as *search()* does. For example, if a writer wanted to find all of the adverbs in some text, they might use *findall()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, *finditer()* is useful as it provides *Match* objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use *finditer()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Writing a Tokenizer

A tokenizer or scanner analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
```

(繼續下一頁)

(繼續上一頁)

```
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

The tokenizer produces the following output:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.4 difflib --- 計算差🄵的輔助工具

原始碼： Lib/difflib.py

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the *filecmp* module.

class difflib.**SequenceMatcher**

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are *hashable*. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements; these "junk" elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershelp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. *SequenceMatcher* is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: *SequenceMatcher* supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item's duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as "popular" and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the `autojunk` argument to `False` when creating the *SequenceMatcher*.

在 3.2 版的變更: 新增 *autojunk* 參數。

class difflib.**Differ**

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses *SequenceMatcher* both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a *Differ* delta begins with a two-letter code:

Code	含義
`'- '`	line unique to sequence 1
`'+ '`	line unique to sequence 2
`'  '`	line common to both sequences
`'? '`	line not present in either input sequence

Lines beginning with '`?`' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain whitespace characters, such as spaces, tabs or line breaks.

class difflib.**HtmlDiff**

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Initializes instance of *HtmlDiff*.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to `8`.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into `ndiff()` (used by `HtmlDiff` to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to `True` when contextual differences are to be shown, else the default is `False` to show the full files. *numlines* defaults to `5`. When *context* is `True` *numlines* controls the number of context lines which surround the difference highlights. When *context* is `False` *numlines* controls the number of lines which are shown before a difference highlight when using the "next" hyperlinks (setting to zero would cause the "next" hyperlinks to place the next difference highlight at the top of the browser without any leading context).

> **備🄵**
>
> *fromdesc* and *todesc* are interpreted as unescaped HTML and should be properly escaped while receiving input from untrusted sources.

在 3.5 版的變更: *charset* keyword-only argument was added. The default charset of HTML document changed from `'ISO-8859-1'` to `'utf-8'`.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the `make_file()` method.

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py',
...                       tofile='after.py'))
*** before.py
--- after.py
```

(繼續下一頁)

(繼續上一頁)

```
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

一個更詳盡的範例請見*difflib 的命令列介面*。

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Compare *a* and *b* (lists of strings); return a *`Differ`*-style delta (a *generator* generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are filtering functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function *`IS_LINE_JUNK()`*, which filters out lines without visible characters, except for at most one pound character (`'#'`) -- however the underlying *`SequenceMatcher`* class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function *`IS_CHARACTER_JUNK()`*, which filters out whitespace characters (a blank or tab; it's a bad idea to include newline in this!).

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
```

(繼續下一頁)

(繼續上一頁)

```
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

Return one of the two sequences that generated a delta.

Given a *sequence* produced by *Differ.compare()* or *ndiff()*, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

範例：

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from *io.IOBase.readlines()* result in diffs that are suitable for use with *io.IOBase.writelines()* since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The unified diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile='after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

一個更詳盡的範例請見*difflib 的命令列介面*。

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

Compare *a* and *b* (lists of bytes objects) using *dfunc*; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either *unified_diff()* or *context_diff()*.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile,`

`tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

在 3.5 版被加入.

`difflib.`**`IS_LINE_JUNK`**(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in *ndiff()* in older versions.

`difflib.`**`IS_CHARACTER_JUNK`**(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in *ndiff()*.

也參考

Pattern Matching: The Gestalt Approach
Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

6.4.1 SequenceMatcher 物件

The *SequenceMatcher* class has this constructor:

`class` `difflib.`**`SequenceMatcher`**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be *hashable*.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

在 3.2 版的變更: 新增 *autojunk* 參數。

SequenceMatcher objects get three data attributes: *bjunk* is the set of elements of *b* for which *isjunk* is `True`; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled); *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with *set_seqs()* or *set_seq2()*.

在 3.2 版被加入: The *bjunk* and *bpopular* attributes.

SequenceMatcher 物件有以下方法:

`set_seqs`(*a*, *b*)

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use *set_seq2()* to set the commonly used sequence once and call *set_seq1()* repeatedly, once for each of the other sequences.

`set_seq1`(*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

`set_seq2`(*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, *find_longest_match()* returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents `' abcd'` from matching the `' abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a *named tuple* `Match(a, b, size)`.

在 3.9 版的變更: 新增預設引數。

get_matching_blocks()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n < i'` or `j+n < j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings:

Value	含義
`'replace'`	`a[i1:i2]` 應該被替🄵🄵 `b[j1:j2]`。
`'delete'`	`a[i1:i2]` 應該被🄵除。請注意，在這種情🄵下 `j1 == j2`。
`'insert'`	`b[j1:j2]` should be inserted at `a[i1:i1]`. Note that `i1 == i2` in this case.
`'equal'`	`a[i1:i2] == b[j1:j2]` (the sub-sequences are equal).

舉例來🄵:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by *get_opcodes()*, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as *get_opcodes()*.

ratio()

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is `1.0` if the sequences are identical, and `0.0` if they have nothing in common.

This is expensive to compute if *get_matching_blocks()* or *get_opcodes()* hasn't already been called, in which case you may want to try *quick_ratio()* or *real_quick_ratio()* first to get an upper bound.

> **備🄵**
>
> Caution: The result of a *ratio()* call may depend on the order of the arguments. For instance:
>
> ```
> >>> SequenceMatcher(None, 'tide', 'diet').ratio()
> 0.25
> >>> SequenceMatcher(None, 'diet', 'tide').ratio()
> 0.5
> ```

quick_ratio()

Return an upper bound on *ratio()* relatively quickly.

real_quick_ratio()

Return an upper bound on *ratio()* very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although *quick_ratio()* and *real_quick_ratio()* are always at least as large as *ratio()*:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.4.2 SequenceMatcher 范例

This example compares two strings, considering blanks to be ”junk”:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a *ratio()* value over 0.6 means the sequences are close matches:

```
>>> print(round(s.ratio(), 3))
0.866
```

If you’re only interested in where the sequences match, *get_matching_blocks()* is handy:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by *get_matching_blocks()* is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is `0`.

If you want to know how to change the first sequence into the second, use *get_opcodes()*:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

也參考

- The *get_close_matches()* function in this module which shows how simple code building on *SequenceMatcher* can be used to do useful work.
- Simple version control recipe for a small application built with *SequenceMatcher*.

6.4.3 Differ Objects

Note that *Differ*-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

Differ 類🄵有這個建構函式:

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the *find_longest_match()* method’s *isjunk* parameter for an explanation.

Differ objects are used (deltas generated) via a single method:

compare(*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the *readlines()* method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the *writelines()* method of a file-like object.

6.4.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the *readlines()* method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a *Differ* object we may pass functions to filter out line and character ”junk.” See the *Differ()* constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let’s pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                     ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
```

(繼續下一頁)

(繼續上一頁)

```
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?          ++++ ^                         ^
+   5. Flat is better than nested.
```

6.4.5 difflib 的命令列介面

This example shows how to use difflib to create a `diff`-like utility.

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
```

(繼續下一頁)

(繼續上一頁)

```
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4.6 ndiff 范例：

This example shows how to use *difflib.ndiff()*.

```
"""ndiff [-q] file1 file2
    or
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

Print a human-friendly file difference report to stdout.  Both inter-
and intra-line differences are noted.  In the second form, recreate file1
(-r1) or file2 (-r2) on stdout, from an ndiff report on stdin.

In the first form, if -q ("quiet") is not specified, the first two lines
of output are

-: file1
+: file2

Each remaining line begins with a two-letter code:

    "- "    line unique to file1
    "+ "    line unique to file2
    "  "    line common to both files
    "? "    line not present in either input file

Lines beginning with "? " attempt to guide the eye to intraline
differences, and were not present in either input file.  These lines can be
confusing if the source files contain tab characters.

The first file can be recovered by retaining only lines that begin with
"  " or "- ", and deleting those 2-character prefixes; use ndiff with -r1.

The second file can be recovered similarly, but by retaining only "  " and
"+ " lines; use ndiff with -r2; or, on Unix, the second file can be
recovered by piping the output through

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
    out(__doc__)
    return 0
```

(繼續下一頁)

(繼續上一頁)

```
# open a file & return the file object; gripe and return 0 if it
# couldn't be opened
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " + str(detail))

# open two files & spray the diff to stdout; return false iff a problem
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
        print(line, end=' ')

    return 1

# crack args (sys.argv[1:] is normal) & compare;
# return false iff a problem

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# read ndiff output from stdin, and print file1 (which=='1') or
# file2 (which=='2') to stdout

def restore(which):
```

(繼續下一頁)

(繼續上一頁)

```
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

6.5 textwrap --- 文字包裝與填充

原始碼： Lib/textwrap.py

The *textwrap* module provides some convenience functions, as well as *TextWrapper*, the class that does all the work. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of *TextWrapper* for efficiency.

textwrap.**wrap**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. Returns a list of output lines, without final newlines.

Optional keyword arguments correspond to the instance attributes of *TextWrapper*, documented below.

See the *TextWrapper.wrap()* method for additional details on how *wrap()* behaves.

textwrap.**fill**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. *fill()* is shorthand for

```
"\n".join(wrap(text, ...))
```

In particular, *fill()* accepts exactly the same keyword arguments as *wrap()*.

textwrap.**shorten**(*text*, *width*, ***, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder=' [...]'*)

Collapse and truncate the given *text* to fit in the given *width*.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the *placeholder* fit within *width*:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Optional keyword arguments correspond to the instance attributes of *TextWrapper*, documented below. Note that the whitespace is collapsed before the text is passed to the *TextWrapper fill()* function, so changing the value of *tabsize*, *expand_tabs*, *drop_whitespace*, and *replace_whitespace* will have no effect.

在 3.4 版被加入.

textwrap.**dedent**(*text*)

Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `" hello"` and `"\thello"` are considered to have no common leading whitespace.

Lines containing only whitespace are ignored in the input and normalized to a single newline character in the output.

舉例來🄵:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

Add *prefix* to the beginning of selected lines in *text*.

Lines are separated by calling `text.splitlines(True)`.

By default, *prefix* is added to all lines that do not consist solely of whitespace (including any line endings).

舉例來🄵:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

The optional *predicate* argument can be used to control which lines are indented. For example, it is easy to add *prefix* to even empty and whitespace-only lines:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

在 3.3 版被加入.

wrap(), *fill()* and *shorten()* work by creating a *TextWrapper* instance and calling a single method on it. That instance is not reused, so for applications that process many text strings using *wrap()* and/or *fill()*, it may be more efficient to create your own *TextWrapper* object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless *TextWrapper.break_long_words* is set to false.

class textwrap.**TextWrapper**(***kwargs*)

The *TextWrapper* constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can reuse the same *TextWrapper* object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The *TextWrapper* instance attributes (and keyword arguments to the constructor) are as follows:

width

(default: `70`) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than *width*, *TextWrapper* guarantees that no output line will be longer than *width* characters.

expand_tabs

(default: `True`) If true, then all tab characters in *text* will be expanded to spaces using the *expandtabs()* method of *text*.

tabsize

(default: `8`) If *expand_tabs* is true, then all tab characters in *text* will be expanded to zero or more spaces, depending on the current column and the given tab size.

在 3.3 版被加入.

replace_whitespace

(default: `True`) If true, after tab expansion but before wrapping, the *wrap()* method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'\t\n\v\f\r'`).

> **備🄵**
>
> If *expand_tabs* is false and *replace_whitespace* is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

> **備🄵**
>
> If *replace_whitespace* is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using *str.splitlines()* or similar) which are wrapped separately.

drop_whitespace

(default: `True`) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

initial_indent

(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent

(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings

(default: `False`) If true, *TextWrapper* attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this algorithm is that it is unable to detect the difference between ”Dr.” in

```
[...] Dr. Frankenstein's monster [...]
```

and ”Spot.” in

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of "lowercase letter", and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words

(default: `True`) If true, then words longer than *width* will be broken in order to ensure that no lines are longer than *width*. If it is false, long words will not be broken, and some lines may be longer than *width*. (Long words will be put on a line by themselves, in order to minimize the amount by which *width* is exceeded.)

break_on_hyphens

(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set *break_long_words* to false if you want truly insecable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

max_lines

(default: `None`) If not `None`, then the output will contain at most *max_lines* lines, with *placeholder* appearing at the end of the output.

在 3.4 版被加入.

placeholder

(default: `' [...]'`) String that will appear at the end of the output text if it has been truncated.

在 3.4 版被加入.

TextWrapper also provides some public methods, analogous to the module-level convenience functions:

wrap(*text*)

Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. All wrapping options are taken from instance attributes of the *TextWrapper* instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(*text*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph.

6.6 `unicodedata` --- Unicode 資料庫

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 16.0.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, "Unicode Character Database". It defines the following functions:

也參考

The unicode-howto for more information about Unicode and how to use this module.

`unicodedata.`**lookup**(*name*)

Look up character by name. If a character with the given name is found, return the corresponding character. If not found, *KeyError* is raised. For example:

```
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
```

The characters returned by this function are the same as those produced by `\N` escape sequence in string literals. For example:

```
>>> unicodedata.lookup('MIDDLE DOT') == '\N{MIDDLE DOT}'
True
```

在 3.3 版的變更: 已新增對名稱🄴名[1] 和命名序列[2] 的支援。

`unicodedata.`**`name`**(*chr*, *default=None*, */*)

Returns the name assigned to the character *chr* as a string. If no name is defined, *default* is returned, or, if not given, *ValueError* is raised. For example:

```
>>> unicodedata.name('½')
'VULGAR FRACTION ONE HALF'
>>> unicodedata.name('\uFFFF', 'fallback')
'fallback'
```

`unicodedata.`**`decimal`**(*chr*, *default=None*, */*)

Returns the decimal value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised. For example:

```
>>> unicodedata.decimal('\N{ARABIC-INDIC DIGIT NINE}')
9
>>> unicodedata.decimal('\N{SUPERSCRIPT NINE}', -1)
-1
```

`unicodedata.`**`digit`**(*chr*, *default=None*, */*)

Returns the digit value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised:

```
>>> unicodedata.digit('\N{SUPERSCRIPT NINE}')
9
```

`unicodedata.`**`numeric`**(*chr*, *default=None*, */*)

Returns the numeric value assigned to the character *chr* as float. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised:

```
>>> unicodedata.numeric('½')
0.5
```

`unicodedata.`**`category`**(*chr*)

Returns the general category assigned to the character *chr* as string. General category names consist of two letters. See the General Category Values section of the Unicode Character Database documentation for a list of category codes. For example:

```
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
```

`unicodedata.`**`bidirectional`**(*chr*)

Returns the bidirectional class assigned to the character *chr* as string. If no such value is defined, an empty string is returned. See the Bidirectional Class Values section of the Unicode Character Database documentation for a list of bidirectional codes. For example:

[1] https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/16.0.0/ucd/NamedSequences.txt

```
>>> unicodedata.bidirectional('\N{ARABIC-INDIC DIGIT SEVEN}') # 'A'rabic, 'N'umber
'AN'
```

unicodedata.**combining**(*chr*)

Returns the canonical combining class assigned to the character *chr* as integer. Returns `0` if no combining class is defined. See the Canonical Combining Class Values section of the Unicode Character Database for more information.

unicodedata.**east_asian_width**(*chr*)

Returns the east asian width assigned to the character *chr* as string. For a list of widths and or more information, see the Unicode Standard Annex #11.

unicodedata.**mirrored**(*chr*)

Returns the mirrored property assigned to the character *chr* as integer. Returns `1` if the character has been identified as a "mirrored" character in bidirectional text, `0` otherwise. For example:

```
>>> unicodedata.mirrored('>')
1
```

unicodedata.**decomposition**(*chr*)

Returns the character decomposition mapping assigned to the character *chr* as string. An empty string is returned in case no such mapping is defined. For example:

```
>>> unicodedata.decomposition('Ã')
'0041 0303'
```

unicodedata.**normalize**(*form*, *unistr*)

Return the normal form *form* for the Unicode string *unistr*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (for example, gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, that is, replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.

unicodedata.**is_normalized**(*form*, *unistr*)

Return whether the Unicode string *unistr* is in the normal form *form*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

在 3.8 版被加入.

In addition, the module exposes the following constant:

unicodedata.**unidata_version**

在此模組中使用的 Unicode 資料庫版本。

unicodedata.**ucd_3_2_0**

This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

6.7 stringprep --- 網際網路字串的準備

原始碼： Lib/stringprep.py

在網際網路上識🄵事物（例如主機名稱）時，通常需要比較這些識🄵器的「相等性 (equality)」。比較的具體執行方式可能取🄵於應用程式領域，例如是否應該不區分大小寫。也可能有必要限制可能的識🄵器，只允許由「可列印 (printable)」字元組成的識🄵器。

RFC 3454 定義了在網際網路通訊協定中「准備」Unicode 字串的程序。在傳送字串到網路之前，先使用准備程序處理字串，之後字串就具有特定的規範化 (normalized) 形式。RFC 定義了一系列的表，這些表可以組合成設定檔 (profile)。每個設定檔必須定義它使用哪些表以及 stringprep 程序的哪些其他可選部分是設定檔的一部分。stringprep 配置文件的一個例子是 nameprep，它被用於國際化網域名稱。

stringprep 模組只開放 **RFC 3454** 中的表。由於這些表如果以字典或串列的方式來表示的話會非常大，因此模組🄵部使用 Unicode 字元資料庫。模組原始碼本身是使用 mkstringprep.py 工具來🄵生的。

因此，這些表是以函式而非資料結構的形式呈現。RFC 中有兩種表：集合 (sets) 和對映 (mappings)。對於集合，*stringprep* 提供了「特徵函式 (characteristic function)」，也就是如果參數是集合的一部分就回傳 True 的函式。對於對映，它提供了對映函式：給定鍵，它會回傳相關的值。以下是模組中所有可用函式的清單。

stringprep.**in_table_a1**(*code*)

判斷 *code* 是否在 tableA.1（Unicode 3.2 中未指定的編碼位置 (code point)）中。

stringprep.**in_table_b1**(*code*)

判斷 *code* 是否在 tableB.1（通常🄵有對映到任何東西）中。

stringprep.**map_table_b2**(*code*)

根據 tableB.2（使用 NFKC 形式的大小寫折🄵 (case-folding) 對映）回傳 *code* 的對映值。

stringprep.**map_table_b3**(*code*)

根據 tableB.3（使用🄵有規範化的大小寫折🄵對映）回傳 *code* 的對映值。

stringprep.**in_table_c11**(*code*)

判斷 *code* 是否在 tableC.1.1（ASCII 空格字元）中。

stringprep.**in_table_c12**(*code*)

判斷 *code* 是否在 tableC.1.2（非 ASCII 空格字元）中。

stringprep.**in_table_c11_c12**(*code*)

判斷 *code* 是否在 tableC.1（空格字元，🄵 C.1.1 和 C.1.2 的聯集）中。

stringprep.**in_table_c21**(*code*)

判斷 *code* 是否在 tableC.2.1（ASCII 控制字元）中。

stringprep.**in_table_c22**(*code*)

判斷 *code* 是否在 tableC.2.2（非 ASCII 控制字元）中。

stringprep.**in_table_c21_c22**(*code*)

判斷 *code* 是否在 tableC.2（控制字元，🄵 C.2.1 和 C.2.2 的聯集）中。

stringprep.**in_table_c3**(*code*)

判斷 *code* 是否在 tableC.3（私有使用）中。

stringprep.**in_table_c4**(*code*)

判斷 *code* 是否在 tableC.4（非字元編碼位置）中。

stringprep.**in_table_c5**(*code*)
判斷 *code* 是否在 tableC.5（代理碼）中。

stringprep.**in_table_c6**(*code*)
判斷 *code* 是否在 tableC.6（不適用於純文字）中。

stringprep.**in_table_c7**(*code*)
判斷 *code* 是否在 tableC.7（不適用於規範表示法 (canonical representation)）中。

stringprep.**in_table_c8**(*code*)
判斷 *code* 是否在 tableC.8（變更顯示屬性或已🄵用）中。

stringprep.**in_table_c9**(*code*)
判斷 *code* 是否在 tableC.9（標記字元 (tagging characters)）中。

stringprep.**in_table_d1**(*code*)
判斷 *code* 是否在 tableD.1（具有雙向屬性"R" 或"AL" 的字元）中。

stringprep.**in_table_d2**(*code*)
判斷 *code* 是否在 tableD.2（具有雙向屬性"L" 的字元）中。

6.8 `readline` --- GNU readline 介面

The *readline* module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the *rlcompleter* module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in *input()* function.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

備🄵

The underlying Readline library API may be implemented by the `editline` (`libedit`) library instead of GNU readline. On macOS the *readline* module detects which library is being used at run time.

The configuration file for `editline` is different from that of GNU readline. If you programmatically load configuration strings you can use *backend* to determine which library is being used.

If you use `editline`/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion:

```
python:bind -v
python:bind ^I rl_complete
```

Also note that different libraries may use different history file formats. When switching the underlying library, existing history files may become unusable.

readline.**backend**
The name of the underlying Readline library being used, either `"readline"` or `"editline"`.

在 3.13 版被加入.

6.8.1 Init file

The following functions relate to the init file and user configuration:

`readline.`**`parse_and_bind`**(*string*)

Execute the init line provided in the *string* argument. This calls `rl_parse_and_bind()` in the underlying library.

`readline.`**`read_init_file`**([*filename*])

Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library. It raises an *auditing event* `open` with the file name if given, and `"<readline_init_file>"` otherwise, regardless of which file the library resolves.

在 3.14 版的變更: The auditing event was added.

6.8.2 Line buffer

The following functions operate on the line buffer:

`readline.`**`get_line_buffer`**()

Return the current contents of the line buffer (`rl_line_buffer` in the underlying library).

`readline.`**`insert_text`**(*string*)

Insert text into the line buffer at the cursor position. This calls `rl_insert_text()` in the underlying library, but ignores the return value.

`readline.`**`redisplay`**()

Change what's displayed on the screen to reflect the current contents of the line buffer. This calls `rl_redisplay()` in the underlying library.

6.8.3 History file

The following functions operate on a history file:

`readline.`**`read_history_file`**([*filename*])

Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls `read_history()` in the underlying library and raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

在 3.14 版的變更: The auditing event was added.

`readline.`**`write_history_file`**([*filename*])

Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls `write_history()` in the underlying library and raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

在 3.14 版的變更: The auditing event was added.

`readline.`**`append_history_file`**(*nelements*[, *filename*])

Append the last *nelements* items of history to a file. The default filename is `~/.history`. The file must already exist. This calls `append_history()` in the underlying library. This function only exists if Python was compiled for a version of the library that supports it. It raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

在 3.5 版被加入.

在 3.14 版的變更: The auditing event was added.

`readline.`**`get_history_length`**()

`readline.`**`set_history_length`**(*length*)

Set or return the desired number of lines to save in the history file. The *`write_history_file()`* function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.

6.8.4 History list

The following functions operate on a global history list:

readline.**clear_history**()

Clear the current history. This calls `clear_history()` in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

readline.**get_current_history_length**()

Return the number of items currently in the history. (This is different from *get_history_length()*, which returns the maximum number of lines that will be written to a history file.)

readline.**get_history_item**(*index*)

Return the current contents of history item at *index*. The item index is one-based. This calls `history_get()` in the underlying library.

readline.**remove_history_item**(*pos*)

Remove history item specified by its position from the history. The position is zero-based. This calls `remove_history()` in the underlying library.

readline.**replace_history_item**(*pos*, *line*)

Replace history item specified by its position with *line*. The position is zero-based. This calls `replace_history_entry()` in the underlying library.

readline.**add_history**(*line*)

Append *line* to the history buffer, as if it was the last line typed. This calls `add_history()` in the underlying library.

readline.**set_auto_history**(*enabled*)

Enable or disable automatic calls to `add_history()` when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

在 3.6 版被加入.

CPython 實作細節： Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.8.5 Startup hooks

readline.**set_startup_hook**([*function*])

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.**set_pre_input_hook**([*function*])

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

6.8.6 Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by *rlcompleter* to complete Python identifiers for the interactive interpreter. If the *readline* module is to be used with a custom completer, a different set of word delimiters should be set.

readline.**set_completer**([*function*])

Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as

`function(text, state)`, for *state* in `0`, `1`, `2`, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

`readline.`**`get_completer`**`()`

Get the completer function, or `None` if no completer function has been set.

`readline.`**`get_completion_type`**`()`

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

`readline.`**`get_begidx`**`()`

`readline.`**`get_endidx`**`()`

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library. The values may be different in the same input editing scenario based on the underlying C readline implementation. Ex: libedit is known to behave differently than libreadline.

`readline.`**`set_completer_delims`**`(`*string*`)`

`readline.`**`get_completer_delims`**`()`

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

`readline.`**`set_completion_display_matches_hook`**`(`[*function*]`)`

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

6.8.7 范例

The following example demonstrates how to use the *readline* module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's `PYTHONSTARTUP` file.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see *Readline configuration*).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```
import atexit
import os
```

(繼續下一頁)

(繼續上一頁)

```
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

The following example extends the *code.InteractiveConsole* class to support history save/restore.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.9 `rlcompleter` --- GNU readline 的補全函式

原始碼： Lib/rlcompleter.py

The `rlcompleter` module defines a completion function suitable to be passed to *set_completer()* in the *readline* module.

When this module is imported on a Unix platform with the *readline* module available, an instance of the *Completer* class is automatically created and its *complete()* method is set as the *readline completer*. The method provides completion of valid Python identifiers and keywords.

範例：

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
```

(繼續下一頁)

(繼續上一頁)

```
>>> readline. <TAB PRESSED>
readline.__doc__           readline.get_line_buffer(  readline.read_init_file(
readline.__file__          readline.insert_text(      readline.set_completer(
readline.__name__          readline.parse_and_bind(
>>> readline.
```

The `rlcompleter` module is designed for use with Python's interactive mode. Unless Python is run with the `-S` option, the module is automatically imported and configured (see *Readline configuration*).

On platforms without *readline*, the *Completer* class defined by this module can still be used for custom purposes.

class rlcompleter.**Completer**

Completer objects have the following method:

complete(*text*, *state*)

Return the next possible completion for *text*.

When called by the *readline* module, this method is called successively with `state == 0, 1, 2, ...` until the method returns `None`.

If called for *text* that doesn't include a period character (`'.'`), it will complete from names currently defined in *__main__*, *builtins* and keywords (as defined by the *keyword* module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to `__getattr__()`) up to the last part, and find matches for the rest via the *dir()* function. Any exception raised during the evaluation of the expression is caught, silenced and *None* is returned.

CHAPTER 7

二進位資料服務

本章所描述的模組提供了一些基本的二進位資料操作服務。而針對二進位資料的其他操作 —— 尤其是關於檔案格式和網路協定的部分 —— 則會在相關章節中詳細描述。

一些在*文本處理 (Text Processing) 服務* 中提及的函式庫也可處理 ASCII 相容的二進位格式（例如，`re`）及所有的二進位資料（例如，`difflib`）。

此外，請參🄵 Python 🄵建的二進位資料類型的文件，詳情請參考*Binary Sequence Types --- bytes, bytearray, memoryview*。

7.1 `struct` --- 將位元組直譯🄵打包起來的二進位資料

原始碼： Lib/struct.py

此模組可在 Python 數值與以 Python `bytes` 物件表示的 C 結構之間進行轉🄴。簡潔的格式字串描述了與 Python 數值之間預期的轉🄵。此模組的函式和物件可用於兩種截然不同的應用程式：與外部來源（檔案或網路連🄵）的資料交🄵，或是 Python 應用程式與 C 層之間的資料傳輸。

> **備🄵**
>
> 當未給定前綴字元時會預設🄵原生模式。它會根據建構 Python 直譯器的平台和編譯器來打包或解包資料。打包給定 C 結構的結果包含填充位元組，以維持相關 C 型🄴的正確對齊；同樣地，解包時也會考量對齊。相反地，當在外部來源之間傳輸資料時，程式設計師需負責定義位元組順序和元素之間的填充。詳情請參見位元組順序、大小和對齊。

一些`struct` 函式（和`Struct` 的方法）接受一個 *buffer* 引數。這是指實作 bufferobjects 🄵提供可讀或可讀寫緩衝區的物件。最常用於此目的的型🄵是`bytes` 和`bytearray`，但許多其他可以視🄵位元組陣列的型🄵都實作了緩衝區協定，因此它們可以從`bytes` 物件讀取或填入資料而無需額外的🄵🄵。

7.1.1 函式與例外

此模組定義了以下例外和函式：

exception `struct.`**error**

在各種情🄴下都可能觸發的例外；引數是一個描述錯誤🄴容的字串。

struct.**pack**(*format*, *v1*, *v2*, ...)

回傳一個包含依照格式字串 *format* 打包的數值 *v1*、*v2*、... 的位元組物件。引數必須完全符合格式所需的數值。

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)

依照格式字串 *format* 打包數值 *v1*、*v2*、...，🄵將打包後的位元組寫入可寫緩衝區 *buffer* 中從位置 *offset* 開始的地方。請注意 *offset* 是必要的引數。

struct.**unpack**(*format*, *buffer*)

根據格式字串 *format* 從緩衝區 *buffer*（推測是由 `pack(format, ...)` 打包的）解包。結果是一個元組，即使它只包含一個項目。緩衝區的位元組大小必須符合格式所需的大小，如*calcsize()* 所示。

struct.**unpack_from**(*format*, */*, *buffer*, *offset=0*)

根據格式字串 *format* 從 *buffer* 的 *offset* 位置開始解包。即使它只包含一個項目，結果也都是一個元組。緩衝區從位置 *offset* 開始的位元組大小，必須至少達到格式所需的大小，如*calcsize()* 所示。

struct.**iter_unpack**(*format*, *buffer*)

根據格式字串 *format* 🄵代地從緩衝區 *buffer* 解包。此函式回傳一個🄵代器，它會從緩衝區讀取等大小的區塊，直到消耗完所有🄵容。緩衝區的位元組大小必須是格式所需大小的倍數，如*calcsize()* 所示。

每次🄵代都會🄵生 (yield) 出一個由格式字串指定的元組。

在 3.4 版被加入.

struct.**calcsize**(*format*)

回傳對應於格式字串 *format* 的結構大小（因此也是 `pack(format, ...)` 所🄵生的位元組物件的大小）。

7.1.2 格式字串

格式字串描述了打包和解包資料時的資料配置 (layout)。它們由格式字元組成，這些字元指定了要打包或解包的資料型🄵。此外，特殊字元控制位元組順序、大小和對齊。每個格式字串由一個可選的前綴字元組成，描述資料的整體屬性，以及一個或多個描述實際資料值和填充的格式字元。

位元組順序、大小和對齊

預設情🄵下，C 型🄵以機器的原生格式和位元組順序表示，🄵在必要時透過跳過填充位元組來正確對齊（根據 C 編譯器所使用的規則）。選擇此行🄵使得打包結構的位元組完全對應於相對應 C 結構的記憶體配置。是否使用原生位元組順序和填充或標準格式取🄵於應用程式。

或者，格式字串的第一個字元可以用來指示打包資料的位元組順序、大小和對齊，如下表所示：

字元	位元組順序	大小	對齊
`@`	原生	原生	原生
`=`	原生	標準	無
`<`	小端序	標準	無
`>`	大端序	標準	無
`!`	網路（= 大端序）	標準	無

如果第一個字元不是這些字元之一，則假設🄵 `'@'`。

備🄵

數字 1023（十六進位🄵 `0x3ff`）具有以下位元組表示法：

- 大端序（`>`）🄵 `03 ff`
- 小端序（`<`）🄵 `ff 03`

Python 範例：

```
>>> import struct
>>> struct.pack('>h', 1023)
b'\x03\xff'
>>> struct.pack('<h', 1023)
b'\xff\x03'
```

原生位元組順序是大端序或小端序，會取🄵於主機系統。例如，Intel x86、AMD64 (x86-64) 和 Apple M1 是小端序；IBM z 和許多舊架構是大端序。可使用 *sys.byteorder* 來檢查你系統的位元組順序。

原生大小和對齊是使用 C 編譯器的 `sizeof` 運算式來🄵定的。這總是與原生位元組順序結合使用。

標准大小僅取🄵於格式字元；請參見格式字元區塊中的表格。

請注意 `'@'` 和 `'='` 之間的差🄵：兩者都使用原生位元組順序，但後者的大小和對齊是標准化的。

`'!'` 形式表示網路位元組順序，根據 IETF RFC 1700 的定義，它總是大端序。

🄵有方法來指示非原生位元組順序（🄵制位元組交🄵）；請使用適當的 `'<'` 或 `'>'` 選擇。

🄵解：

(1) 填充只會在連續的結構成員之間自動加入。編碼結構的開頭或結尾不會加入填充。

(2) 使用非原生大小和對齊時，例如使用'<'、'>'、'=' 和'!' 時，不會加入填充。

(3) 要將結構的結尾對齊到特定型🄵的對齊需求，請以該型🄵的碼結束格式，重🄵次數🄵零。請參見範例。

格式字元

格式字元具有以下意義；在給定型🄵的情🄵下，C 和 Python 數值之間的轉🄵應該是顯而易見的。「標准大小」欄位指的是使用標准大小時（即當格式字串以 `'<'`、`'>'`、`'!'` 或 `'='` 其中之一開始時）打包數值的位元組大小。使用原生大小時，打包數值的大小取🄵於平台。

格式	C Type	Python 型🄵	標准大小	🄵解
x	填充位元組	無值		(7)
c	char	長度🄵 1 的位元組	1	
b	signed char	整數	1	(1), (2)
B	unsigned char	整數	1	(2)
?	_Bool	bool	1	(1)
h	short	整數	2	(2)
H	unsigned short	整數	2	(2)
i	int	整數	4	(2)
I	unsigned int	整數	4	(2)
l	long	整數	4	(2)
L	unsigned long	整數	4	(2)
q	long long	整數	8	(2)
Q	unsigned long long	整數	8	(2)
n	ssize_t	整數		(3)
N	size_t	整數		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
F	float complex	complex	8	(10)
D	double complex	complex	16	(10)
s	char[]	位元組		(9)
p	char[]	位元組		(8)
P	void*	整數		(5)

在 3.3 版的變更: 新增 'n' 與 'N' 格式的支援。

在 3.6 版的變更: 新增 'e' 格式的支援。

在 3.14 版的變更: 新增 'F' 與 'D' 格式的支援。

🄵解:

(1) '?' 轉🄴碼對應於自 C99 以來 C 標準定義的 _Bool 型🄵。在標準模式下，它由一個位元組表示。

(2) 當嘗試使用任何整數轉🄴碼打包非整數時，如果非整數具有 __index__() 方法，則會呼叫該方法在打包之前將引數轉🄵🄵整數。

在 3.2 版的變更: 新增對非整數使用 __index__() 方法的支援。

(3) 'n' 和 'N' 轉🄴碼僅適用於原生大小（作🄵預設選擇或使用 '@' 位元組順序字元）。對於標準大小，你可以使用適合你應用程式的其他整數格式。

(4) 對於 'f'、'd' 和 'e' 轉🄴碼，打包表示法使用 IEEE 754 binary32、binary64 或 binary16 格式（分🄵對應於 'f'、'd' 或 'e'），無論平台使用何種浮點數格式。

(5) 'P' 格式字元僅適用於原生位元組順序（作🄵預設選擇或使用 '@' 位元組順序字元）。位元組順序字元 '=' 根據主機系統選擇使用小端序或大端序。struct 模組不將其解釋🄵原生順序，因此 'P' 格式不可用。

(6) IEEE 754 binary16「半精度 (half precision)」型🄵是在 2008 年的 IEEE 754 標准修訂版中引入的。它有一個符號位元 (sign bit)、5 位元指數和 11 位元精度（明確儲存 10 位元），可以全精度表示大約 6.1e-05 到 6.5e+04 之間的數字。此型🄵未被 C 編譯器廣泛支援：在典型機器上，unsigned short 可以用於儲存但不能用於數學運算。請參見 Wikipedia 上的半精度浮點數格式頁面以取得更多資訊。

(7) 打包時，'x' 插入一個 NUL 位元組。

(8) 'p' 格式字元編碼一個「Pascal 字串」，意思是儲存在固定位元組數中的短、長度可變字串，由計數指定。儲存的第一個位元組是字串的長度，或與 255 間取較小者。字串的位元組在其後。如果傳遞給*pack()* 的字串太長（長於計數🄴 1），則只儲存字串的前 count-1 個位元組。如果字串短於 count-1，則用空位元組填充，使得總共使用恰好 count 個位元組。請注意，對於*unpack()*，'p' 格式字元消耗 count 個位元組，但回傳的字串永遠不能包含超過 255 個位元組。

(9) 對於 's' 格式字元，計數被直譯🄴位元組的長度，而不是像其他格式字元那樣的重🄴次數；例如，'10s' 表示一個 10 位元組字串，對應於單一 Python 位元組字串，而 '10c' 表示 10 個獨立的單位元組字元元素（例如 cccccccccc），對應於十個不同的 Python 位元組物件。（請參見範例 以了解具體差🄵的示範。）如果未給定計數，則預設🄴 1。打包時，會適當地截斷或用空位元組填充字串以使其適合。解包時，結果位元組物件總是恰好具有指定的位元組數。作🄴特殊情🄵，'0s' 表示單一空字串（而 '0c' 表示 0 個字元）。

(10) For the 'F' and 'D' format characters, the packed representation uses the IEEE 754 binary32 and binary64 format for components of the complex number, regardless of the floating-point format used by the platform. Note that complex types (F and D) are available unconditionally, despite complex types being an optional feature in C. As specified in the C11 standard, each complex type is represented by a two-element C array containing, respectively, the real and imaginary parts.

格式字元前面可以加上整數重🄴次數。例如，格式字串 '4h' 與 'hhhh' 意思完全相同。

格式之間的空白字元會被忽略；但是計數和其格式不能包含空白字元。

當使用整數格式之一（'b'、'B'、'h'、'H'、'i'、'I'、'l'、'L'、'q'、'Q'）打包數值 x 時，如果 x 超出該格式的有效範圍，則會觸發*struct.error*。

在 3.1 版的變更: 以前一些整數格式會環繞超出範圍的數值🄵引發*DeprecationWarning* 而不是*struct.error*。

對於 '?' 格式字元，回傳值是*True* 或*False*。打包時，使用引數物件的真值。原生或標準布林值表示法中的 0 或 1 將被打包，解包時任何非零值都將🄴 True。

范例

> **備🄵**
>
> 原生位元組順序範例（由 '@' 格式前綴或缺少任何前綴字元指定）可能與讀者機器🄵生的結果不符，因🄵這取🄴於平台和編譯器。

使用大端序打包和解包三種不同大小的整數：

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

嘗試打包對於定義欄位來🄵太大的整數：

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

示範 's' 和 'c' 格式字元之間的差🄵：

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

解包的欄位可以透過將其指派給變數或將結果包裝在具名元組中來命名：

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

在原生模式下，格式字元的順序可能會影響大小，因🄵填充是隱式的。在標准模式下，使用者負責插入任何所需的填充。請注意下面第一個 pack 呼叫中，在打包的 '#' 之後加入了三個 NUL 位元組，以將後續整數對齊到四位元組邊界。在此範例中，輸出是在小端序機器上🄴生的：

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

假設平台的 long 對齊到 4 位元組邊界，以下格式 '`llh0l`' 會在結尾加入兩個填充位元組：

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

也參考

array 模組
: 同質資料的打包二進位儲存。

json 模組
: JSON 編碼器和解碼器。

pickle 模組
: Python 物件序列化。

7.1.3 應用

struct 模組有兩種主要應用程式：在應用程式🄴或使用相同編譯器編譯的另一個應用程式之間進行 Python 和 C 程式碼的資料交🄴（*原生格式*），以及使用約定資料配置的應用程式之間的資料交🄴（*標準格式*）。一般來🄴，🄴這兩個領域建構的格式字串是不同的。

原生格式

當構造模擬原生配置的格式字串時，編譯器和機器架構會🄵定位元組順序和填充。在這種情🄵下，應該使用 `@` 格式字元來指定原生位元組順序和資料大小。🄴部填充位元組通常會自動插入。可能需要在格式字串的結尾使用零重🄴格式碼來向上舍入到正確的位元組邊界，以便正確對齊連續的資料區塊 (chunks)。

考慮這兩個簡單範例（在 64 位元小端序機器上）：

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

在第二個格式字串的結尾，如果不使用額外填充，資料不會填充到 8 位元組邊界。零重🄵格式碼解🄵了這個問題：

```
>>> calcsize('@llh0l')
24
```

`'x'` 格式碼可以用來指定重🄵，但對於原生格式來🄵最好使用像 `'0l'` 這樣的零重🄵格式。

預設使用原生位元組順序和對齊，但最好明確使用 `'@'` 前綴字元。

標準格式

當與你的行程之外的網路或儲存區等交🄴資料時，要🄴精確。指定確切的位元組順序、大小和對齊。不要假設它們與特定機器的原生順序匹配。例如，網路位元組順序是大端序，而許多流行的 CPU 是小端序。透過明確定義這一點，使用者無需關心其程式碼運行的平台細節。第一個字元通常應該是 < 或 >（或 !）。填充是程式設計師的責任。零重🄴格式字元不會起作用。取而代之的是使用者必須在需要的地方明確加入 `'x'` 填充位元組。重新檢視上一節的範例，我們有：

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
```

(繼續下一頁)

(繼續上一頁)

```
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

上述結果（在 64 位元機器上執行）和在不同機器上執行時不保證會匹配。例如，下面的範例是在 32 位元機器上執行的：

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 類🄵

struct 模組也定義了以下型🄵：

class struct.**Struct**(*format*)

回傳一個新的 Struct 物件，它會根據格式字串 *format* 寫入和讀取二進位資料。建立一次 `Struct` 物件🄵呼叫其方法比使用相同格式呼叫模組層級函式更有效率，因🄵格式字串只編譯一次。

> **備🄵**
>
> 傳遞給模組層級函式的最近一個格式字串的編譯版本會被快取起來，因此僅使用少數格式字串的程式無需擔心重🄵使用單一*Struct* 實例。

編譯過的 Struct 物件支援以下方法和屬性：

pack(*v1*, *v2*, ...)

與*pack()* 函式相同，會使用編譯過的格式。(`len(result)` 將等於*size*。)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

與*pack_into()* 函式相同，會使用編譯過的格式。

unpack(*buffer*)

與*unpack()* 函式相同，會使用編譯過的格式。緩衝區的位元組大小必須等於*size*。

unpack_from(*buffer*, *offset=0*)

與*unpack_from()* 函式相同，會使用編譯過的格式。緩衝區從位置 *offset* 開始的位元組大小必須至少🄵*size*。

iter_unpack(*buffer*)

與*iter_unpack()* 函式相同，會使用編譯過的格式。緩衝區的位元組大小必須是*size* 的倍數。

在 3.4 版被加入.

format

用於建構此 Struct 物件的格式字串。

在 3.7 版的變更: 格式字串型🄵現在是*str* 而不是*bytes*。

size

對應於*format* 的結構之計算出的大小（因此也是*pack()* 方法所🄵生的位元組物件的大小）。

在 3.13 版的變更: 結構的 *repr()* 已經改變。現在是：

```
>>> Struct('i')
Struct('i')
```

7.2 codecs --- 編解碼器FF表和基底類F

原始碼： Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with *text encodings* or with codecs that encode to `bytes`.

這個模組定義了以下函式，用於以任何編解碼器來編碼與解碼：

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

Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

`codecs.`**`decode`**(*obj*, *encoding='utf-8'*, *errors='strict'*)

Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that decoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeDecodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

`codecs.`**`charmap_build`**(*string*)

Return a mapping suitable for encoding with a custom single-byte encoding. Given a `str` *string* of up to 256 characters representing a decoding table, returns either a compact internal mapping object `EncodingMap` or a `dictionary` mapping character ordinals to byte values. Raises a `TypeError` on invalid input.

The full details for each codec can also be looked up directly:

`codecs.`**`lookup`**(*encoding*, */*)

Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no `CodecInfo` object is found, a `LookupError` is raised. Otherwise, the `CodecInfo` object is stored in the cache and returned to the caller.

`class` `codecs.`**`CodecInfo`**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

`name`

編碼的名稱。

`encode`

`decode`

The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the `encode()` and `decode()` methods of Codec instances (see *Codec Interface*). The functions or methods are expected to work in a stateless mode.

`incrementalencoder`

`incrementaldecoder`

Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes `IncrementalEncoder` and `IncrementalDecoder`, respectively. Incremental codecs can maintain state.

`streamwriter`

streamreader

Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes *StreamWriter* and *StreamReader*, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use *lookup()* for the codec lookup:

codecs.**getencoder**(*encoding*)

Look up the codec for the given encoding and return its encoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getdecoder**(*encoding*)

Look up the codec for the given encoding and return its decoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getincrementalencoder**(*encoding*)

Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental encoder.

codecs.**getincrementaldecoder**(*encoding*)

Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental decoder.

codecs.**getreader**(*encoding*)

Look up the codec for the given encoding and return its *StreamReader* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getwriter**(*encoding*)

Look up the codec for the given encoding and return its *StreamWriter* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

codecs.**register**(*search_function*, */*)

Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters with hyphens and spaces converted to underscores, and return a *CodecInfo* object. In case a search function cannot find a given encoding, it should return `None`.

在 3.9 版的變更: Hyphens and spaces are converted to underscore.

codecs.**unregister**(*search_function*, */*)

Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing.

在 3.10 版被加入.

While the builtin *open()* and the associated *io* module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

Open an encoded file using the given *mode* and return an instance of *StreamReaderWriter*, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

> **備🄵**
>
> If *encoding* is not `None`, then the underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in *open()* function; the `'b'` is automatically added.

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a *ValueError* to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in *open()* function. It defaults to -1 which means that the default buffer size will be used.

在 3.11 版的變更: 已移除 `'U'` 模式。

在 3.14 版之後被🄵用: *codecs.open()* has been superseded by *open()*.

`codecs.`**`EncodedFile`**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

Return a *StreamRecoder* instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes *ValueError* to be raised in case an encoding error occurs.

`codecs.`**`iterencode`**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental encoder to iteratively encode the input provided by *iterator*. *iterator* must yield *str* objects. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text *str* objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

`codecs.`**`iterdecode`**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental decoder to iteratively decode the input provided by *iterator*. *iterator* must yield *bytes* objects. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept *bytes* objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with *iterencode()*.

`codecs.`**`readbuffer_encode`**(*buffer*, *errors=None*, */*)

Return a *tuple* containing the raw bytes of *buffer*, a buffer-compatible object or *str* (encoded to UTF-8 before processing), and their length in bytes.

忽略 *errors* 引數。

```
>>> codecs.readbuffer_encode(b"Zito")
(b'Zito', 4)
```

The module also provides the following constants which are useful for reading and writing to platform dependent files:

`codecs.`**`BOM`**

`codecs.`**`BOM_BE`**

`codecs.`**`BOM_LE`**

codecs.**BOM_UTF8**

codecs.**BOM_UTF16**

codecs.**BOM_UTF16_BE**

codecs.**BOM_UTF16_LE**

codecs.**BOM_UTF32**

codecs.**BOM_UTF32_BE**

codecs.**BOM_UTF32_LE**

These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. *BOM_UTF16* is either *BOM_UTF16_BE* or *BOM_UTF16_LE* depending on the platform's native byte order, *BOM* is an alias for *BOM_UTF16*, *BOM_LE* for *BOM_UTF16_LE* and *BOM_BE* for *BOM_UTF16_BE*. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.2.1 Codec Base Classes

The *codecs* module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument:

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python 標準編碼 codecs:

Value	含義
`'strict'`	Raise *UnicodeError* (or a subclass), this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in *ignore_errors()*.
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in *replace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

The following error handlers are only applicable to encoding (within *text encodings*):

Value	含義
`'xmlcharref`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;`. Implemented in *xmlcharrefreplace_errors()*.
`'namereplac`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in *namereplace_errors()*.

In addition, the following error handler is specific to the given codecs:

Value	Codecs	含義
`'surrog`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in *str* as an error.

在 3.1 版被加入: The `'surrogateescape'` and `'surrogatepass'` error handlers.

在 3.4 版的變更: The `'surrogatepass'` error handler now works with utf-16* and utf-32* codecs.

在 3.5 版被加入: The `'namereplace'` error handler.

在 3.5 版的變更: The `'backslashreplace'` error handler now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

`codecs.`**`register_error`**(*name*, *error_handler*, /)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

Decoding and translating works similarly, except *UnicodeDecodeError* or *UnicodeTranslateError* will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

`codecs.`**`lookup_error`**(*name*, /)

Return the error handler previously registered under the name *name*.

Raises a *LookupError* in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

`codecs.`**`strict_errors`**(*exception*)

Implements the `'strict'` error handling.

Each encoding or decoding error raises a *UnicodeError*.

`codecs.`**`ignore_errors`**(*exception*)

Implements the `'ignore'` error handling.

Malformed data is ignored; encoding or decoding is continued without further notice.

`codecs.`**`replace_errors`**(*exception*)

Implements the `'replace'` error handling.

Substitutes `?` (ASCII character) for encoding errors or `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

codecs.**backslashreplace_errors**(*exception*)

Implements the `'backslashreplace'` error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use the hexadecimal form of byte value with format `\xhh`.

在 3.5 版的變更: Works with decoding and translating.

codecs.**xmlcharrefreplace_errors**(*exception*)

Implements the `'xmlcharrefreplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` .

codecs.**namereplace_errors**(*exception*)

Implements the `'namereplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}` .

在 3.5 版被加入.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

class codecs.**Codec**

encode(*input*, *errors='strict'*)

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamWriter` for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

decode(*input*, *errors='strict'*)

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface -- for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamReader` for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the *encode()*/*decode()* method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder 物件

The *IncrementalEncoder* class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalEncoder**(*errors='strict'*)

IncrementalEncoder 的建構函式。

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalEncoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalEncoder* object.

encode(*object*, *final=False*)

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to *encode()* *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by *getstate()*.

IncrementalDecoder 物件

The *IncrementalDecoder* class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalDecoder**(*errors='strict'*)

IncrementalDecoder 的建構函式。

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalDecoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalDecoder* object.

decode(*object*, *final=False*)

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to *decode()* *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()

將解碼器重設到初始狀態。

getstate()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that `0` is the most common additional state info.) If this additional state info is `0` it must be possible to set the decoder to the state which has no input buffered and `0` as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the decoder to *state*. *state* must be a decoder state returned by *getstate()*.

串流編碼和解碼

The *StreamWriter* and *StreamReader* classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

StreamWriter 物件

The *StreamWriter* class is a subclass of *Codec* and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

StreamWriter 的建構函式。

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The *StreamWriter* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *StreamWriter* object.

write(*object*)

Writes the object's contents encoded to the stream.

writelines(*list*)

Writes the concatenated iterable of strings to the stream (possibly by reusing the *write()* method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()

Resets the codec buffers used for keeping internal state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the *StreamWriter* must also inherit all other methods and attributes from the underlying stream.

StreamReader 物件

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.**StreamReader**(*stream*, *errors='strict'*)

`StreamReader` 的建構函式。

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read(*size=-1*, *chars=-1*, *firstline=False*)

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline(*size=None*, *keepends=True*)

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines(*sizehint=None*, *keepends=True*)

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()

Resets the codec buffers used for keeping internal state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter 物件

The *StreamReaderWriter* is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the *lookup()* function to construct the instance.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

Creates a *StreamReaderWriter* instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the *StreamReader* and *StreamWriter* interface resp. Error handling is done in the same way as defined for the stream readers and writers.

StreamReaderWriter instances define the combined interfaces of *StreamReader* and *StreamWriter* classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder 物件

The *StreamRecoder* translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the *lookup()* function to construct the instance.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

Creates a *StreamRecoder* instance which implements a two-way conversion: *encode* and *decode* work on the frontend —the data visible to code calling *read()* and *write()*, while *Reader* and *Writer* work on the backend —the data in *stream*.

You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

stream 引數必須是類檔案物件。

The *encode* and *decode* arguments must adhere to the *Codec* interface. *Reader* and *Writer* must be factory functions or classes providing objects of the *StreamReader* and *StreamWriter* interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of *StreamReader* and *StreamWriter* classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 編碼和 Unicode

Strings are stored internally as sequences of code points in range `U+0000`--`U+10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

The simplest text encoding (called `'latin-1'` or `'iso-8859-1'`) maps the code points 0--255 to the bytes `0x0`--`0xff`, which means that a string object that contains code points above `U+00FF` can't be encoded with this codec. Doing so will raise a *UnicodeEncodeError* that looks like the following (although the details of the error message may differ): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).`

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes `0x0`--`0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you

will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in a `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

There's another encoding that is able to encode the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
`U-00000000` ... `U-0000007F`	0xxxxxxx
`U-00000080` ... `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` ... `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` ... `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 標準編碼

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases because they are equivalent when normalized by *`normalize_encoding()`*. For example, `'utf-8'` is a valid alias for the `'utf_8'` codec.

備🄵

The below table lists the most common aliases, for a complete list refer to the source aliases.py file.

On Windows, `cpXXX` codecs are available for all code pages. But only codecs listed in the following table are guaranteed to exist on other platforms.

CPython 實作細節： Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

在 3.6 版的變更: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	語言
ascii	646, us-ascii	英文
big5	big5-tw, csbig5	繁體中文
big5hkscs	big5-hkscs, hkscs	繁體中文
cp037	IBM037, IBM039	英文
cp273	273, IBM273, csIBM273	德文 在 3.4 版被加入.
cp424	EBCDIC-CP-HE, IBM424	希伯來文
cp437	437, IBM437	英文
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		阿拉伯文
cp737		希臘文
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Belarusian, Bulgarian, Macedonian, Russian, Serbian
cp856		希伯來文
cp857	857, IBM857	土耳其文
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	希伯來文
cp863	863, IBM863	Canadian
cp864	IBM864	阿拉伯文
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	俄羅斯文
cp869	869, CP-GR, IBM869	希臘文
cp874		泰文
cp875		希臘文

繼續下一頁

表格 1 – 繼續上一頁

Codec	Aliases	語言
cp932	932, ms932, mskanji, ms-kanji, windows-31j	日文
cp949	949, ms949, uhc	韓文
cp950	950, ms950	繁體中文
cp1006		Urdu
cp1026	ibm1026	土耳其文
cp1125	1125, ibm1125, cp866u, ruscii	烏克蘭文 在 3.4 版被加入.
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Belarusian, Bulgarian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	希臘文
cp1254	windows-1254	土耳其文
cp1255	windows-1255	希伯來文
cp1256	windows-1256	阿拉伯文
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	越南文
euc_jp	eucjp, ujis, u-jis	日文
euc_jis_2004	jisx0213, eucjis2004	日文
euc_jisx0213	eucjisx0213	日文
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韓文
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	簡體中文
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	簡體中文
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日文
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日文
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日文
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日文
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日文
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韓文
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Western Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Belarusian, Bulgarian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	阿拉伯文
iso8859_7	iso-8859-7, greek, greek8	希臘文
iso8859_8	iso-8859-8, hebrew	希伯來文
iso8859_9	iso-8859-9, latin5, L5	土耳其文
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	泰語

繼續下一頁

表格 1 – 繼續上一頁

Codec	Aliases	語言
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	韓文
koi8_r		俄羅斯文
koi8_t		Tajik 在 3.5 版被加入.
koi8_u		烏克蘭文
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh 在 3.5 版被加入.
mac_cyrillic	maccyrillic	Belarusian, Bulgarian, Macedonian, Russian, Serbian
mac_greek	macgreek	希臘文
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	Central and Eastern Europe
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	土耳其文
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日文
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日文
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日文
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages
utf_16_le	UTF-16LE	all languages
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8, cp65001	all languages
utf_8_sig		all languages

在 3.4 版的變更: The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800--U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

在 3.8 版的變更: `cp65001` 現在是 `utf_8` 的🄵名。

在 3.14 版的變更: On Windows, `cpXXX` codecs are now available for all code pages.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide *`str`* to *`bytes`* encoding and *bytes-like object* to *`str`* decoding, similar to the Unicode text encodings.

Codec	Aliases	含義
idna		Implement **RFC 3490**, see also *encodings.idna*. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only: Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only: Encode the operand according to the OEM codepage (CP_OEMCP). 在 3.6 版被加入.
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with `\uXXXX` and `\UXXXXXXXX` for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		This Codec should only be used for testing purposes. Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.

在 3.8 版的變更: "unicode_internal" codec is removed.

Binary Transforms

The following codecs provide binary transforms: *bytes-like object* to `bytes` mappings. They are not supported by `bytes.decode()` (which only produces `str` output).

Codec	Aliases	含義	Encoder / decoder
base64_codec[1]	base64, base_64	Convert the operand to multiline MIME base64 (the result always includes a trailing `'\n'`). 在 3.4 版的變更: accepts any *bytes-like object* as input for encoding and decoding	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	Compress the operand using bz2.	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quotedprint-able, quoted_printal	Convert the operand to MIME quoted printable.	*quopri.encode()* with `quotetabs=True` / *quopri.decode()*
uu_codec	uu	Convert the operand using uuencode.	
zlib_codec	zip, zlib	Compress the operand using gzip.	*zlib.compress()* / *zlib.decompress()*

在 3.2 版被加入: Restoration of the binary transforms.

在 3.4 版的變更: Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform: a *str* to *str* mapping. It is not supported by *str.encode()* (which only produces *bytes* output).

Codec	Aliases	含義
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

在 3.2 版被加入: Restoration of the `rot_13` text transform.

在 3.4 版的變更: Restoration of the `rot13` alias.

7.2.5 `encodings` --- Encodings package

This module implements the following functions:

`encodings.`**`normalize_encoding`**(*encoding*)

Normalize encoding name *encoding*.

Normalization works as follows: all non-alphanumeric characters except the dot used for Python package names are collapsed and replaced with a single underscore, leading and trailing underscores are removed. For example, `' -;#'` becomes `'_'`.

Note that *encoding* should be ASCII only.

備🄵

The following functions should not be used directly, except for testing purposes; *codecs.lookup()* should be used instead.

[1] In addition to *bytes-like objects*, `'base64_codec'` also accepts ASCII-only instances of *str* for decoding

encodings.**search_function**(*encoding*)

Search for the codec module corresponding to the given encoding name *encoding*.

This function first normalizes the *encoding* using *normalize_encoding()*, then looks for a corresponding alias. It attempts to import a codec module from the encodings package using either the alias or the normalized name. If the module is found and defines a valid `getregentry()` function that returns a *codecs.CodecInfo* object, the codec is cached and returned.

If the codec module defines a `getaliases()` function any returned aliases are registered for future use.

encodings.**win32_code_page_search_function**(*encoding*)

Search for a Windows code page encoding *encoding* of the form `cpXXXX`.

If the code page is valid and supported, return a *codecs.CodecInfo* object for it.

可用性: Windows.

在 3.14 版被加入.

This module implements the following exception:

exception encodings.**CodecRegistryError**

Raised when a codec is invalid or incompatible.

7.2.6 `encodings.idna` --- Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and *stringprep*.

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party idna module.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *http.client* and *ftplib*, accept Unicode host names (*http.client* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: applications wishing to present such host names to the user should decode them to Unicode.

The module *encodings.idna* also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

encodings.idna.**nameprep**(*label*)

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

encodings.idna.**ToASCII**(*label*)

Convert a label to ASCII, as specified in **RFC 3490**. `UseSTD3ASCIIRules` is assumed to be false.

encodings.idna.**ToUnicode**(*label*)

Convert a label to Unicode, as specified in **RFC 3490**.

7.2.7 `encodings.mbcs` --- Windows ANSI 碼頁

這個模組實作了 ANSI 碼頁 (CP_ACP)。

可用性: Windows.

在 3.2 版的變更: Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

在 3.3 版的變更: Support any error handler.

7.2.8 `encodings.utf_8_sig` --- UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

CHAPTER 8

資料型🄵

本章節所描述的模組 (module) 提供了多樣的專門資料型🄵，例如日期與時間、固定型🄵陣列 (fixed-type arrays)、堆積🄵列 (heap queues)、雙端🄵列 (double-ended queues) 與列舉 (enumerations)。

Python 也有提供一些🄵建資料型🄵，特🄵是 *dict*、*list*、*set* 與 *frozenset* 和 *tuple*。*str* 類🄵是用來儲存 Unicode 字串，*bytes* 與 *bytearray* 類🄵則是用來儲存二進位制資料。

本章節包含下列模組的文件：

8.1 datetime --- 日期與時間的基本型🄵

原始碼： Lib/datetime.py

`datetime` 模組提供操作日期與時間的類🄵。

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

小訣竅

跳轉至*格式碼 (format codes)*。

也參考

***calendar* 模組**
與日🄵相關的一般函式。

***time* 模組**
Time access and conversions.

***zoneinfo* 模組**
Concrete time zones representing the IANA time zone database.

Package dateutil
帶有時區與剖析擴充支援的第三方函式庫。

DateType 套件
Third-party library that introduces distinct static types to e.g. allow *static type checkers* to differentiate between naive and aware datetimes.

8.1.1 Aware and Naive Objects

Date and time objects may be categorized as ”aware” or ”naive” depending on whether or not they include time zone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents a specific moment in time that is not open to interpretation.[1]

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other time zone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, *datetime* and *time* objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract *tzinfo* class. These *tzinfo* objects capture information about the offset from UTC time, the time zone name, and whether daylight saving time is in effect.

Only one concrete *tzinfo* class, the *timezone* class, is supplied by the `datetime` module. The `timezone` class can represent simple time zones with fixed offsets from UTC, such as UTC itself or North American EST and EDT time zones. Supporting time zones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

8.1.2 常數

`datetime` 模組匯出以下常數:

`datetime.`**`MINYEAR`**
The smallest year number allowed in a *date* or *datetime* object. *MINYEAR* is 1.

`datetime.`**`MAXYEAR`**
The largest year number allowed in a *date* or *datetime* object. *MAXYEAR* is 9999.

`datetime.`**`UTC`**
Alias for the UTC time zone singleton *datetime.timezone.utc*.

在 3.11 版被加入.

8.1.3 Available Types

`class` `datetime.`**`date`**
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: *year*, *month*, and *day*.

`class` `datetime.`**`time`**
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of ”leap seconds” here.) Attributes: *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

`class` `datetime.`**`datetime`**
A combination of a date and a time. Attributes: *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

[1] 也就是🄵，我們會忽略相對論的效應

class datetime.**timedelta**

表示兩個*datetime* 或*date* 實例之間時間的差🄴，以微秒🄴解析度。

class datetime.**tzinfo**

An abstract base class for time zone information objects. These are used by the *datetime* and *time* classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.**timezone**

A class that implements the *tzinfo* abstract base class as a fixed offset from the UTC.

在 3.2 版被加入.

Objects of these types are immutable.

Subclass relationships:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

常見屬性

The *date*, *datetime*, *time*, and *timezone* types share these common features:

- Objects of these types are immutable.
- Objects of these types are *hashable*, meaning that they can be used as dictionary keys.
- Objects of these types support efficient pickling via the *pickle* module.

Determining if an Object is Aware or Naive

Objects of the *date* type are always naive.

An object of type *time* or *datetime* may be aware or naive.

A *datetime* object `d` is aware if both of the following hold:

1. `d.tzinfo` 不是 `None`
2. `d.tzinfo.utcoffset(d)` 不會回傳 `None`

否則 `d` 會是 naive 的。

A *time* object `t` is aware if both of the following hold:

1. `t.tzinfo` 不是 `None`
2. `t.tzinfo.utcoffset(None)` 🄵有回傳 `None`。

否則 `t` 會是 naive 的。

The distinction between aware and naive doesn't apply to *timedelta* objects.

8.1.4 `timedelta` 物件

一個*timedelta* 物件代表著一段持續時間，即兩個*datetime* 或*date* 之間的差🄴。

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- 一毫秒會被轉𝔽𝔽 1000 微秒。
- 一分鐘會被轉𝔽𝔽 60 秒。
- 一小時會被轉𝔽𝔽 3600 秒。
- 一𝔽會被轉𝔽𝔽 7 天。

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

The following example illustrates how any arguments besides *days*, *seconds* and *microseconds* are "merged" and normalized into those three resulting attributes:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, *`OverflowError`* is raised.

Note that normalization of negative values may be surprising at first. For example:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Since the string representation of `timedelta` objects can be confusing, use the following recipe to produce a more readable format:

```
>>> def pretty_timedelta(td):
...     if td.days >= 0:
...         return str(td)
...     return f'-({-td!s})'
...
>>> d = timedelta(hours=-1)
>>> str(d)  # not human-friendly
'-1 day, 23:00:00'
>>> pretty_timedelta(d)
'-(1:00:00)'
```

類🄵屬性:

`timedelta.`**`min`**

The most negative *`timedelta`* object, `timedelta(-999999999)`.

`timedelta.`**`max`**

The most positive *`timedelta`* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.`**`resolution`**

The smallest possible difference between non-equal *`timedelta`* objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` is greater than `-timedelta.min`. `-timedelta.max` is not representable as a *`timedelta`* object.

Instance attributes (read-only):

`timedelta.`**`days`**

在 -999,999,999 到 999,999,999 （含）之間

`timedelta.`**`seconds`**

在 0 到 86,399 （含）之間

警示

It is a somewhat common bug for code to unintentionally use this attribute when it is actually intended to get a *`total_seconds()`* value instead:

```
>>> from datetime import timedelta
>>> duration = timedelta(seconds=11235813)
>>> duration.days, duration.seconds
(130, 3813)
>>> duration.total_seconds()
11235813.0
```

`timedelta.`**`microseconds`**

在 0 到 999,999 （含）之間

Supported operations:

Operation	Result
`t1 = t2 + t3`	Sum of `t2` and `t3`. Afterwards `t1 - t2 == t3` and `t1 - t3 == t2` are true. (1)
`t1 = t2 - t3`	Difference of `t2` and `t3`. Afterwards `t1 == t2 - t3` and `t2 == t1 + t3` are true. (1)(6)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer. Afterwards `t1 // i == t2` is true, provided `i != 0`.
	In general, `t1 * i == t1 * (i-1) + t1` is true. (1)
`t1 = t2 * f or t1 = f * t2`	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`f = t2 / t3`	Division (3) of overall duration `t2` by interval unit `t3`. Returns a *float* object.
`t1 = t2 / f or t1 = t2 / i`	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	The remainder is computed as a *timedelta* object. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. `q` is an integer and `r` is a *timedelta* object.
`+t1`	Returns a *timedelta* object with the same value. (2)
`-t1`	等價於 `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`，也等價於 `t1 * -1`。(1)(4)
`abs(t)`	Equivalent to `+t` when `t.days >= 0`, and to `-t` when `t.days < 0`. (2)
`str(t)`	Returns a string in the form `[D day[s], ][H]H:MM:SS[.UUUUUU]`, where D is negative for negative `t`. (5)
`repr(t)`	Returns a string representation of the *timedelta* object as a constructor call with canonical attribute values.

🄵解：

(1) 這是精確的，但可能會溢位。

(2) 這是精確的，且不會溢位。

(3) Division by zero raises *ZeroDivisionError*.

(4) `-timedelta.max` is not representable as a *timedelta* object.

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when t3 is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

In addition to the operations listed above, *timedelta* objects support certain additions and subtractions with *date* and *datetime* objects (see below).

在 3.2 版的變更: Floor division and true division of a *timedelta* object by another *timedelta* object are now supported, as are remainder operations and the *divmod()* function. True division and multiplication of a *timedelta* object by a *float* object are now supported.

timedelta objects support equality and order comparisons.

In Boolean contexts, a *timedelta* object is considered to be true if and only if it isn't equal to `timedelta(0)`.

實例方法：

`timedelta.`**`total_seconds`**`()`

Return the total number of seconds contained in the duration. Equivalent to `td /`

`timedelta(seconds=1)`. For interval units other than seconds, use the division form directly (e.g. `td / timedelta(microseconds=1)`).

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

在 3.2 版被加入.

用法范例：`timedelta`

An additional example of normalization:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Examples of *`timedelta`* arithmetic:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 `date` 物件

A *`date`* object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on.[2]

class `datetime.`**`date`**(*year*, *month*, *day*)

All arguments are required. Arguments must be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, *`ValueError`* is raised.

Other constructors, all class methods:

classmethod `date.`**`today`**()

回傳目前的本地日期。

這等同於 `date.fromtimestamp(time.time())`。

[2] This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book *Calendrical Calculations*, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

classmethod date.**fromtimestamp**(*timestamp*)

Return the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*.

This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` function, and *OSError* on `localtime()` failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*.

在 3.3 版的變更: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `localtime()` function. Raise *OSError* instead of *ValueError* on `localtime()` failure.

classmethod date.**fromordinal**(*ordinal*)

Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.

ValueError is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date `d`, `date.fromordinal(d.toordinal()) == d`.

classmethod date.**fromisoformat**(*date_string*)

Return a *date* corresponding to a *date_string* given in any valid ISO 8601 format, with the following exceptions:

1. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
2. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
3. Ordinal dates are not currently supported (`YYYY-OOO`).

範例:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

在 3.7 版被加入.

在 3.11 版的變更: Previously, this method only supported the format `YYYY-MM-DD`.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

Return a *date* corresponding to the ISO calendar date specified by year, week and day. This is the inverse of the function *date.isocalendar()*.

在 3.8 版被加入.

classmethod date.**strptime**(*date_string*, *format*)

Return a *date* corresponding to *date_string*, parsed according to *format*. This is equivalent to:

```
date(*(time.strptime(date_string, format)[0:3]))
```

ValueError is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. See also *strftime() 與 strptime() 的行為🄵* and *date.fromisoformat()*.

備🄵

If *format* specifies a day of month without a year a *DeprecationWarning* is emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import date
>>> date_string = "02/29"
>>> when = date.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap year bug.
>>> when.strftime("%B %d")
'February 29'
```

在 3.14 版被加入.

類🄵屬性:

`date.`**`min`**

The earliest representable date, `date(MINYEAR, 1, 1)`.

`date.`**`max`**

The latest representable date, `date(MAXYEAR, 12, 31)`.

`date.`**`resolution`**

The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

`date.`**`year`**

Between *MINYEAR* and *MAXYEAR* inclusive.

`date.`**`month`**

在 1 到 12 （含）之間。

`date.`**`day`**

Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
`date2 = date1 + timedelta`	`date2` will be `timedelta.days` days after `date1`. (1)
`date2 = date1 - timedelta`	Computes `date2` such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	Equality comparison. (4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
`date1 >= date2`
	Order comparison. (5)

🄵解:

(1) *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. *OverflowError* is raised if `date2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*.

(2) `timedelta.seconds` 和 `timedelta.microseconds` 被忽略。

(3) This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.

(4) *`date`* objects are equal if they represent the same date.

`date` objects that are not also *`datetime`* instances are never equal to `datetime` objects, even if they represent the same date.

(5) *date1* is considered less than *date2* when *date1* precedes *date2* in time. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`.

Order comparison between a `date` object that is not also a *`datetime`* instance and a `datetime` object raises *`TypeError`*.

在 3.13 版的變更: Comparison between *`datetime`* object and an instance of the *`date`* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

In Boolean contexts, all *`date`* objects are considered to be true.

實例方法:

`date.`**`replace`**(*year=self.year*, *month=self.month*, *day=self.day*)

Return a new *`date`* object with the same values, but with specified parameters updated.

範例:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

The generic function *`copy.replace()`* also supports *`date`* objects.

`date.`**`timetuple`**()

回傳一個*`time.struct_time`*，如同*`time.localtime()`* 所回傳。

The hours, minutes and seconds are 0, and the DST flag is -1.

`d.timetuple()` 等價於:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st.

`date.`**`toordinal`**()

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any *`date`* object `d`, `date.fromordinal(d.toordinal()) == d`.

`date.`**`weekday`**()

回傳一個代表星期幾的整數，星期一🄵 0、星期日🄵 6。例如 `date(2002, 12, 4).weekday() == 2` 🄵星期三。也請參考*`isoweekday()`*。

`date.`**`isoweekday`**()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also *`weekday()`*, *`isocalendar()`*.

`date.`**`isocalendar`**()

Return a *named tuple* object with three components: `year`, `week` and `weekday`.

The ISO calendar is a widely used variant of the Gregorian calendar.[3]

[3] See R. H. van Gent's guide to the mathematics of the ISO 8601 calendar for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

在 3.9 版的變更: Result changed from a tuple to a *named tuple*.

`date.`**`isoformat`**`()`

回傳一以 ISO 8601 格式 `YYYY-MM-DD` 表示的日期字串:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

`date.`**`__str__`**`()`

For a date `d`, `str(d)` is equivalent to `d.isoformat()`.

`date.`**`ctime`**`()`

Return a string representing the date:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

`d.ctime()` 等價於:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C `ctime()` function (which *`time.ctime()`* invokes, but which *`date.ctime()`* does not invoke) conforms to the C standard.

`date.`**`strftime`**`(`*format*`)`

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See also *strftime() 與 strptime() 的行🄵* and *`date.isoformat()`*.

`date.`**`__format__`**`(`*format*`)`

Same as *`date.strftime()`*. This makes it possible to specify a format string for a *`date`* object in formatted string literals and when using *`str.format()`*. See also *strftime() 與 strptime() 的行🄵* and *`date.isoformat()`*.

用法范例: `date`

計算一個事件的天數的範例:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
```

(繼續下一頁)

(繼續上一頁)

```
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

更多*date*的用法範例:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 `datetime` 物件

A *datetime* object is a single object containing all the information from a *date* object and a *time* object.

Like a *date* object, *datetime* assumes the current Gregorian calendar extended in both directions; like a *time* object, *datetime* assumes there are exactly 3600*24 seconds in every day.

建構函式:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The *year*, *month* and *day* arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass.

The remaining arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= number of days in the given month and year`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`。

If an argument outside those ranges is given, *ValueError* is raised.

在 3.6 版的變更: 新增 *fold* 參數。

Other constructors, all class methods:

classmethod datetime.**today**()

回傳目前的本地日期與時間，且*tzinfo* 🄵 None。

等價於:

```
datetime.fromtimestamp(time.time())
```

也請見*now()*、*fromtimestamp()*。

This method is functionally equivalent to *now()*, but without a `tz` parameter.

classmethod datetime.**now**(*tz=None*)

Return the current local date and time.

如果選用的引數 *tz* 🄵 None 或未指定，則會像是*today()*，但盡可能提供比透過*time.time()* 取得的時間戳記更多位數的資訊（例如，這在有提供 C `gettimeofday()` 函式的平台上可能可行）。

If *tz* is not None, it must be an instance of a *tzinfo* subclass, and the current date and time are converted to *tz*’s time zone.

This function is preferred over *today()* and *utcnow()*.

備🄵

Subsequent calls to `datetime.now()` may return the same instant depending on the precision of the underlying clock.

classmethod datetime.**utcnow**()

Return the current UTC date and time, with *tzinfo* None.

This is like *now()*, but returns the current UTC date and time, as a naive *datetime* object. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also *now()*.

警告

Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing the current time in UTC is by calling `datetime.now(timezone.utc)`.

在 3.12 版之後被🄵用: Use *datetime.now()* with *UTC* instead.

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

Return the local date and time corresponding to the POSIX timestamp, such as is returned by *time.time()*. If optional argument *tz* is None or not specified, the timestamp is converted to the platform's local date and time, and the returned *datetime* object is naive.

If *tz* is not None, it must be an instance of a *tzinfo* subclass, and the timestamp is converted to *tz*' s time zone.

fromtimestamp() may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions, and *OSError* on localtime() or gmtime() failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*, and then it's possible to have two timestamps differing by a second that yield identical *datetime* objects. This method is preferred over *utcfromtimestamp()*.

在 3.3 版的變更: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions. Raise *OSError* instead of *ValueError* on localtime() or gmtime() failure.

在 3.6 版的變更: *fromtimestamp()* may return instances with *fold* set to 1.

classmethod datetime.**utcfromtimestamp**(*timestamp*)

Return the UTC *datetime* corresponding to the POSIX timestamp, with *tzinfo* None. (The resulting object is naive.)

This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C gmtime() function, and *OSError* on gmtime() failure. It's common for this to be restricted to years in 1970 through 2038.

To get an aware *datetime* object, call *fromtimestamp()*:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

except the latter formula always supports the full years range: between *MINYEAR* and *MAXYEAR* inclusive.

> **⚠ 警告**
>
> Because naive datetime objects are treated by many datetime methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing a specific timestamp in UTC is by calling datetime.fromtimestamp(timestamp, tz=timezone.utc).

在 3.3 版的變更: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C gmtime() function. Raise *OSError* instead of *ValueError* on gmtime() failure.

在 3.12 版之後被🄴用: Use *datetime.fromtimestamp()* with *UTC* instead.

classmethod datetime.**fromordinal**(*ordinal*)

Return the *datetime* corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless 1 <= ordinal <= datetime.max.toordinal(). The hour, minute, second and microsecond of the result are all 0, and *tzinfo* is None.

classmethod datetime.**combine**(*date*, *time*, *tzinfo=time.tzinfo*)

Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components are equal to the given *time* object's. If the *tzinfo* argument is provided, its value is used to set the *tzinfo* attribute of the result, otherwise the *tzinfo* attribute of the *time* argument is used. If the *date* argument is a *datetime* object, its time components and *tzinfo* attributes are ignored.

For any *datetime* object `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

在 3.6 版的變更: 新增 *tzinfo* 引數。

classmethod datetime.**fromisoformat**(*date_string*)

Return a *datetime* corresponding to a *date_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The `T` separator may be replaced by any single unicode character.
3. Fractional hours and minutes are not supported.
4. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
5. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
6. Ordinal dates are not currently supported (`YYYY-OOO`).

範例:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

在 3.7 版被加入.

在 3.11 版的變更: Previously, this method only supported formats that could be emitted by *date.isoformat()* or *datetime.isoformat()*.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

Return a *datetime* corresponding to the ISO calendar date specified by year, week and day. The non-date components of the datetime are populated with their normal default values. This is the inverse of the function *datetime.isocalendar()*.

在 3.8 版被加入.

classmethod datetime.**strptime**(*date_string*, *format*)

Return a *datetime* corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or time zone information, this is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

ValueError is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. See also *strftime() 與 strptime() 的行*🄵 and *datetime.fromisoformat()*.

在 3.13 版的變更: If *format* specifies a day of month without a year a *DeprecationWarning* is now emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python

3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap year bug.
>>> when.strftime("%B %d")
'February 29'
```

類🄵屬性:

`datetime.`**`min`**

The earliest representable *`datetime`*, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

`datetime.`**`max`**

The latest representable *`datetime`*, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

`datetime.`**`resolution`**

The smallest possible difference between non-equal *`datetime`* objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

`datetime.`**`year`**

Between *`MINYEAR`* and *`MAXYEAR`* inclusive.

`datetime.`**`month`**

在 1 到 12（含）之間。

`datetime.`**`day`**

Between 1 and the number of days in the given month of the given year.

`datetime.`**`hour`**

In `range(24)`.

`datetime.`**`minute`**

In `range(60)`.

`datetime.`**`second`**

In `range(60)`.

`datetime.`**`microsecond`**

In `range(1000000)`.

`datetime.`**`tzinfo`**

The object passed as the *tzinfo* argument to the *`datetime`* constructor, or `None` if none was passed.

`datetime.`**`fold`**

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

在 3.6 版被加入.

Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 == datetime2` `datetime1 != datetime2`	Equality comparison. (4)

`datetime1 < datetime2`
`datetime1 > datetime2`
`datetime1 <= datetime2`
`datetime1 >= datetime2`
	Order comparison. (5)

(1) `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same *`tzinfo`* attribute as the input datetime, and `datetime2 - datetime1 == timedelta` after. *`OverflowError`* is raised if `datetime2.year` would be smaller than *`MINYEAR`* or larger than *`MAXYEAR`*. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same *`tzinfo`* attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

(3) Subtraction of a *`datetime`* from a *`datetime`* is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, *`TypeError`* is raised.

If both are naive, or both are aware and have the same *`tzinfo`* attribute, the *`tzinfo`* attributes are ignored, and the result is a *`timedelta`* object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *`tzinfo`* attributes, `a-b` acts as if `a` and `b` were first converted to naive UTC datetimes. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *`datetime`* objects are equal if they represent the same date and time, taking into account the time zone.

Naive and aware `datetime` objects are never equal.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *`fold`* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *`tzinfo`* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. `datetime` instances in a repeated interval are never equal to `datetime` instances in other time zone.

(5) *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time, taking into account the time zone.

Order comparison between naive and aware *`datetime`* objects raises *`TypeError`*.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *`fold`* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *`tzinfo`* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

在 3.3 版的變更: Equality comparisons between aware and naive *`datetime`* instances don't raise *`TypeError`*.

在 3.13 版的變更: Comparison between *`datetime`* object and an instance of the *`date`* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

實例方法:

datetime.**date**()

Return *date* object with same year, month and day.

datetime.**time**()

Return *time* object with same hour, minute, second, microsecond and fold. *tzinfo* is None. See also method *timetz()*.

在 3.6 版的變更: The fold value is copied to the returned *time* object.

datetime.**timetz**()

Return *time* object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method *time()*.

在 3.6 版的變更: The fold value is copied to the returned *time* object.

datetime.**replace**(*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

Return a new *datetime* object with the same attributes, but with specified parameters updated. Note that tzinfo=None can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

datetime objects are also supported by generic function *copy.replace()*.

在 3.6 版的變更: 新增 *fold* 參數。

datetime.**astimezone**(*tz=None*)

Return a *datetime* object with new *tzinfo* attribute *tz*, adjusting the date and time data so the result is the same UTC time as *self*, but in *tz*'s local time.

If provided, *tz* must be an instance of a *tzinfo* subclass, and its *utcoffset()* and *dst()* methods must not return None. If *self* is naive, it is presumed to represent time in the system time zone.

If called without arguments (or with tz=None) the system local time zone is assumed for the target time zone. The .tzinfo attribute of the converted datetime instance will be set to an instance of *timezone* with the zone name and offset obtained from the OS.

If self.tzinfo is *tz*, self.astimezone(tz) is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the time zone *tz*, representing the same UTC time as *self*: after astz = dt.astimezone(tz), astz - astz.utcoffset() will have the same date and time data as dt - dt.utcoffset().

If you merely want to attach a *timezone* object *tz* to a datetime *dt* without adjustment of date and time data, use dt.replace(tzinfo=tz). If you merely want to remove the timezone object from an aware datetime *dt* without conversion of date and time data, use dt.replace(tzinfo=None).

Note that the default *tzinfo.fromutc()* method can be overridden in a *tzinfo* subclass to affect the result returned by *astimezone()*. Ignoring error cases, *astimezone()* acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

在 3.3 版的變更: *tz* now can be omitted.

在 3.6 版的變更: The *astimezone()* method can now be called on naive instances that are presumed to represent system local time.

datetime.**utcoffset**()

If *tzinfo* is None, returns None, else returns self.tzinfo.utcoffset(self), and raises an exception if the latter doesn't return None or a *timedelta* object with magnitude less than one day.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

datetime.**dst**()

If *tzinfo* is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None or a *timedelta* object with magnitude less than one day.

在 3.7 版的變更: The DST offset is not restricted to a whole number of minutes.

datetime.**tzname**()

If *tzinfo* is None, returns None, else returns self.tzinfo.tzname(self), raises an exception if the latter doesn't return None or a string object,

datetime.**timetuple**()

回傳一個*time.struct_time*，如同*time.localtime()* 所回傳。

d.timetuple() 等價於：

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The *tm_isdst* flag of the result is set according to the *dst()* method: *tzinfo* is None or *dst()* returns None, tm_isdst is set to -1; else if *dst()* returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.**utctimetuple**()

If *datetime* instance d is naive, this is the same as d.timetuple() except that *tm_isdst* is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If d is aware, d is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that an *OverflowError* may be raised if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

⚠ 警告

Because naive datetime objects are treated by many datetime methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using *datetime.utctimetuple()* may give misleading results. If you have a naive datetime representing UTC, use datetime.replace(tzinfo=timezone.utc) to make it aware, at which point you can use *datetime.timetuple()*.

datetime.**toordinal**()

Return the proleptic Gregorian ordinal of the date. The same as self.date().toordinal().

datetime.**timestamp**()

Return POSIX timestamp corresponding to the *datetime* instance. The return value is a *float* similar to that returned by *time.time()*.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C mktime() function to perform the conversion. Since *datetime* supports wider range of values than mktime() on many platforms, this method may raise *OverflowError* or *OSError* for times far in the past or far in the future.

For aware *datetime* instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

在 3.3 版被加入.

在 3.6 版的變更: The *timestamp()* method uses the *fold* attribute to disambiguate the times during a repeated interval.

備F

There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system time zone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

or by calculating the timestamp directly:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

`datetime.`**`weekday`**`()`

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also *isoweekday()*.

`datetime.`**`isoweekday`**`()`

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also *weekday()*, *isocalendar()*.

`datetime.`**`isocalendar`**`()`

Return a *named tuple* with three components: `year`, `week` and `weekday`. The same as `self.date().isocalendar()`.

`datetime.`**`isoformat`**`(`*sep='T'*, *timespec='auto'*`)`

Return a string representing the date and time in ISO 8601 format:

- `YYYY-MM-DDTHH:MM:SS.ffffff`，如果*microsecond* 不是 0
- `YYYY-MM-DDTHH:MM:SS`，如果*microsecond* 是 0

如果*utcoffset()* 没有回傳 `None`，則會附加一个字串，給出 UTC 偏移:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`，如果*microsecond* 不是 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`，如果*microsecond* 是 0

範例:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
```

(繼續下一頁)

(繼續上一頁)

```
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit `HH` format.
- `'minutes'`: Include *hour* and *minute* in `HH:MM` format.
- `'seconds'`: Include *hour*, *minute*, and *second* in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

> **備🄵**
>
> Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

在 3.6 版的變更: 新增 *timespec* 參數。

`datetime.__str__()`

For a *datetime* instance `d`, `str(d)` is equivalent to `d.isoformat(' ')`.

`datetime.ctime()`

Return a string representing the date and time:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

The output string will *not* include time zone information, regardless of whether the input is aware or naive.

`d.ctime()` 等價於:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *datetime.ctime()* does not invoke) conforms to the C standard.

`datetime.strftime(format)`

Return a string representing the date and time, controlled by an explicit format string. See also *strftime() 與 strptime() 的行🄵* and *datetime.isoformat()*.

`datetime.__format__(format)`

Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object in formatted string literals and when using *str.format()*. See also *strftime() 與 strptime() 的行🄵* and *datetime.isoformat()*.

用法范例：datetime

更多*datetime*的用法範例：

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", "month
↪", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

The example below defines a *tzinfo* subclass capturing time zone information for Kabul, Afghanistan, which used +4 UTC until 1945 and then +4:30 UTC thereafter:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)
```

(繼續下一頁)

(繼續上一頁)

```
    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

Usage of `KabulTz` from above:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 `time` 物件

A *time* object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

All arguments are optional. *tzinfo* may be None, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1]。

If an argument outside those ranges is given, *ValueError* is raised. All default to 0 except *tzinfo*, which defaults to None.

類🄵屬性:

time.**min**

The earliest representable *time*, time(0, 0, 0, 0).

time.**max**

The latest representable *time*, time(23, 59, 59, 999999).

time.**resolution**

The smallest possible difference between non-equal *time* objects, timedelta(microseconds=1), although note that arithmetic on *time* objects is not supported.

Instance attributes (read-only):

time.**hour**

In range(24).

time.**minute**

In range(60).

time.**second**

In range(60).

time.**microsecond**

In range(1000000).

time.**tzinfo**

The object passed as the tzinfo argument to the *time* constructor, or None if none was passed.

time.**fold**

In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

在 3.6 版被加入.

time objects support equality and order comparisons, where a is considered less than b when a precedes b in time.

Naive and aware time objects are never equal. Order comparison between naive and aware time objects raises *TypeError*.

If both comparands are aware, and have the same *tzinfo* attribute, the tzinfo and fold attributes are ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()).

在 3.3 版的變更: Equality comparisons between aware and naive *time* instances don't raise *TypeError*.

In Boolean contexts, a *time* object is always considered to be true.

在 3.5 版的變更: Before Python 3.5, a *time* object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

其他建構函式:

classmethod time.**fromisoformat**(*time_string*)

Return a *time* corresponding to a *time_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The leading `T`, normally required in cases where there may be ambiguity between a date and a time, is not required.
3. Fractional seconds may have any number of digits (anything beyond 6 will be truncated).
4. Fractional hours and minutes are not supported.

範例:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

在 3.7 版被加入.

在 3.11 版的變更: Previously, this method only supported formats that could be emitted by *time.isoformat()*.

classmethod time.**strptime**(*date_string*, *format*)

Return a *time* corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or timezone information, this is equivalent to:

```
time(*(time.strptime(date_string, format)[3:6]))
```

ValueError is raised if the *date_string* and *format* cannot be parsed by *time.strptime()* or if it returns a value which is not a time tuple. See also *strftime() 與 strptime() 的行為🄵* and *time.fromisoformat()*.

在 3.14 版被加入.

實例方法:

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)

Return a new *time* with the same values, but with specified parameters updated. Note that `tzinfo=None` can be specified to create a naive *time* from an aware *time*, without conversion of the time data.

time objects are also supported by generic function *copy.replace()*.

在 3.6 版的變更: 新增 *fold* 參數。

time.**isoformat**(*timespec='auto'*)

Return a string representing the time in ISO 8601 format, one of:

- `HH:MM:SS.ffffff`, if *microsecond* is not 0
- `HH:MM:SS`, if *microsecond* is 0
- `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if *utcoffset()* does not return `None`
- `HH:MM:SS+HH:MM[:SS[.ffffff]]`, if *microsecond* is 0 and *utcoffset()* does not return `None`

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit `HH` format.
- `'minutes'`: Include *hour* and *minute* in `HH:MM` format.
- `'seconds'`: Include *hour*, *minute*, and *second* in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

> **備🄵**
>
> Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument.

範例:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec='minutes
↪')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

在 3.6 版的變更: 新增 *timespec* 參數。

time.**__str__**()

For a time `t`, `str(t)` is equivalent to `t.isoformat()`.

time.**strftime**(*format*)

Return a string representing the time, controlled by an explicit format string. See also *strftime() 與 strptime() 的行🄵* and *time.isoformat()*.

time.**__format__**(*format*)

Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object in formatted string literals and when using *str.format()*. See also *strftime() 與 strptime() 的行🄵* and *time.isoformat()*.

time.**utcoffset**()

If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a *timedelta* object with magnitude less than one day.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

time.**dst**()

If *tzinfo* is None, returns None, else returns self.tzinfo.dst(None), and raises an exception if the latter doesn't return None, or a *timedelta* object with magnitude less than one day.

在 3.7 版的變更: The DST offset is not restricted to a whole number of minutes.

time.**tzname**()

If *tzinfo* is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception if the latter doesn't return None or a string object.

用法范例：time

Examples of working with a *time* object:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 tzinfo 物件

class datetime.**tzinfo**

This is an abstract base class, meaning that this class should not be instantiated directly. Define a subclass of *tzinfo* to capture information about a particular time zone.

An instance of (a concrete subclass of) *tzinfo* can be passed to the constructors for *datetime* and *time* objects. The latter objects view their attributes as being in local time, and the *tzinfo* object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The datetime module provides *timezone*, a simple concrete subclass of *tzinfo* which can represent time zones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling: A *tzinfo* subclass must have an __init__() method that can be called with no arguments, otherwise it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of *tzinfo* may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware datetime objects. If in doubt, simply implement all of them.

tzinfo.**utcoffset**(*dt*)

Return offset of local time from UTC, as a *timedelta* object that is positive east of UTC. If local time is west of UTC, this should be negative.

This represents the *total* offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If *utcoffset()* does not return `None`, *dst()* should not return `None` either.

The default implementation of *utcoffset()* raises *NotImplementedError*.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

tzinfo.**dst**(*dt*)

Return the daylight saving time (DST) adjustment, as a *timedelta* object or `None` if DST information isn't known.

Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the *tm_isdst* flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

An instance *tz* of a *tzinfo* subclass that models both standard and daylight times must be consistent in this sense:

`tz.utcoffset(dt) - tz.dst(dt)`

must return the same result for every *datetime* *dt* with `dt.tzinfo == tz`. For sane *tzinfo* subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of *datetime.astimezone()* relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a *tzinfo* subclass cannot guarantee this, it may be able to override the default implementation of *tzinfo.fromutc()* to work correctly with *astimezone()* regardless.

Most implementations of *dst()* will probably look like one of these two:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

或是:

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of *dst()* raises *NotImplementedError*.

在 3.7 版的變更: The DST offset is not restricted to a whole number of minutes.

tzinfo.**tzname**(*dt*)

Return the time zone name corresponding to the *datetime* object *dt*, as a string. Nothing about string names is defined by the datetime module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return None if a string name isn't known. Note that this is a method rather than a fixed string primarily because some *tzinfo* subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the *tzinfo* class is accounting for daylight time.

The default implementation of *tzname()* raises *NotImplementedError*.

These methods are called by a *datetime* or *time* object, in response to their methods of the same names. A *datetime* object passes itself as the argument, and a *time* object passes None as the argument. A *tzinfo* subclass's methods should therefore be prepared to accept a *dt* argument of None, or of class *datetime*.

When None is passed, it's up to the class designer to decide the best response. For example, returning None is appropriate if the class wishes to say that time objects don't participate in the *tzinfo* protocols. It may be more useful for utcoffset(None) to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a *datetime* object is passed in response to a *datetime* method, dt.tzinfo is the same object as *self*. *tzinfo* methods can rely on this, unless user code calls *tzinfo* methods directly. The intent is that the *tzinfo* methods interpret *dt* as being in local time, and not need worry about objects in other time zones.

There is one more *tzinfo* method that a subclass may wish to override:

tzinfo.**fromutc**(*dt*)

This is called from the default *datetime.astimezone()* implementation. When called from that, dt.tzinfo is *self*, and *dt*'s date and time data are to be viewed as expressing a UTC time. The purpose of *fromutc()* is to adjust the date and time data, returning an equivalent datetime in *self*'s local time.

Most *tzinfo* subclasses should be able to inherit the default *fromutc()* implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default *fromutc()* implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of *astimezone()* and *fromutc()* may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default *fromutc()* implementation acts like:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

In the following tzinfo_examples.py file there are some examples of *tzinfo* classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)
```

(繼續下一頁)

(繼續上一頁)

```
# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt
```

(繼續下一頁)

(繼續上一頁)

```
# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)
```

(繼續下一頁)

(繼續上一頁)

```
    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a *tzinfo* subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the ”start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM

doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. *`astimezone()`* mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the *`fold`* attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the *`datetime`* instances that differ only by the value of the *`fold`* attribute are considered equal in comparisons.

Applications that can't bear wall-time ambiguities should explicitly check the value of the *`fold`* attribute or avoid using hybrid *`tzinfo`* subclasses; there are no ambiguities when using *`timezone`*, or any other fixed-offset *`tzinfo`* subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

也參考

zoneinfo
: The `datetime` module has a basic *`timezone`* class (for handling arbitrary fixed offsets from UTC) and its *`timezone.utc`* attribute (a UTC `timezone` instance).

  `zoneinfo` brings the *IANA time zone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA 時區資料庫
: The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

8.1.9 timezone 物件

The *timezone* class is a subclass of *tzinfo*, each instance of which represents a time zone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent time zone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

The *offset* argument must be specified as a *timedelta* object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise *ValueError* is raised.

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the *datetime.tzname()* method.

在 3.2 版被加入.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

timezone.**utcoffset**(*dt*)

Return the fixed value specified when the *timezone* instance is constructed.

The *dt* argument is ignored. The return value is a *timedelta* instance equal to the difference between the local time and UTC.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

timezone.**tzname**(*dt*)

Return the fixed value specified when the *timezone* instance is constructed.

If *name* is not provided in the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is "UTC", otherwise it is a string in the format `UTC±HH:MM`, where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

在 3.6 版的變更: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'UTC+00:00'`.

timezone.**dst**(*dt*)

總是回傳 `None`。

timezone.**fromutc**(*dt*)

Return `dt + offset`. The *dt* argument must be an aware *datetime* instance, with `tzinfo` set to `self`.

類🄵屬性:

timezone.**utc**

UTC 時區, `timezone(timedelta(0))`。

8.1.10 strftime() 與 strptime() 的行🄵

date, *datetime*, and *time* objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string.

Conversely, the *date.strptime()*, *datetime.strptime()* and *time.strptime()* class methods create an object from a string representing the time and a corresponding format string.

The table below provides a high-level comparison of *strftime()* versus *strptime()*:

	strftime	strptime
用法	Convert object to a string according to a given format	Parse a string into an object given a corresponding format
Type of method	實例方法	類🄵方法
Signature	`strftime(format)`	`strptime(date_string, format)`

strftime() 與 strptime() 格式碼

These methods accept format codes that can be used to parse and format dates:

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

The following is a list of all the format codes that the 1989 C standard requires, and these work on all platforms with a standard C implementation.

Directive	含義	範例	F解
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	(9)
%b	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	以零填充的F以十進位數字表示的月份。	01, 02, ..., 12	(9)
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99	(9)
%Y	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	(9)
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	(9)
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59	(9)
%S	Second as a zero-padded decimal number.	00, 01, ..., 59	(4), (9)
%f	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, ..., 999999	(5)
%z	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive).	(empty), +0000, -0400, +1030, +063415, -030712.345216	(6)
%Z	Time zone name (empty	(empty), UTC, GMT	(6)

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values.

Directive	含義	範例	🄵解
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (`%V`).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7	
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, 02, ..., 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

These may not be available on all platforms when used with the *`strftime()`* method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling *`strptime()`* with incomplete or ambiguous ISO 8601 directives will raise a *`ValueError`*.

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *`strftime(3)`* documentation. There are also differences between platforms in handling of unsupported format specifiers.

在 3.6 版被加入: 新增 `%G`、`%u` 與 `%V`。

在 3.12 版被加入: 新增 `%:z`。

技術細節

Broadly speaking, `d.strftime(fmt)` acts like the *`time`* module's `time.strftime(fmt, d.timetuple())` although not all objects support a *`timetuple()`* method.

For the *`datetime.strptime()`* class method, the default value is `1900-01-01T00:00:00.000`: any components not specified in the format string will be pulled from the default value.[4]

Using `datetime.strptime(date_string, format)` is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or time zone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

對*`time`* 物件來🄵，不應該使用年、月、日的格式碼，因🄵 `time` 物件🄵🄵有這些值。如果使用這些格式碼，年份會以 1900 代替、月及日會以 1 代替。

對*`date`* 物件來🄵，不應該使用時、分、秒、微秒的格式碼，因🄵*`date`* 物件🄵🄵有這些值。如果使用這些格式碼，這些值都會以 0 代替。

For the same reason, handling of format strings containing Unicode code points that can't be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others `strftime` may raise *`UnicodeError`* or return an empty string instead.

🄵解:

(1) Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain non-ASCII characters.

(2) The *`strptime()`* method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

在 3.2 版的變更: In previous versions, *`strftime()`* method was restricted to years >= 1900.

[4] Passing `datetime.strptime('Feb 29', '%b %d')` will fail since 1900 is not a leap year.

在 3.3 版的變更: In version 3.2, *strftime()* method was restricted to years >= 1000.

(3) When used with the *strptime()* method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) Unlike the *time* module, the `datetime` module does not support leap seconds.

(5) When used with the *strptime()* method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) For a naive object, the `%z`, `%:z` and `%Z` format codes are replaced by empty strings.

For an aware object:

%z

utcoffset() is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if *utcoffset()* returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

在 3.7 版的變更: The UTC offset is not restricted to a whole number of minutes.

在 3.7 版的變更: When the `%z` directive is provided to the *strptime()* method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

%:z

Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

%Z

In *strftime()*, `%Z` is replaced by an empty string if *tzname()* returns `None`; otherwise `%Z` is replaced by the returned value, which must be a string.

strptime() only accepts certain values for `%Z`:

1. any value in `time.tzname` for your machine's locale
2. the hard-coded values `UTC` and `GMT`

So someone living in Japan may have `JST`, `UTC`, and `GMT` as valid values, but probably not `EST`. It will raise `ValueError` for invalid values.

在 3.2 版的變更: When the `%z` directive is provided to the *strptime()* method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) When used with the *strptime()* method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

(8) Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a *strptime()* format string. Also note that `%G` and `%Y` are not interchangeable.

(9) When used with the *strptime()* method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

(10) When parsing a month and day using *strptime()*, always include a year in the format. If the value you need to parse lacks a year, append an explicit dummy leap year. Otherwise your code will raise an exception when it encounters leap day because the default year used by the parser is not a leap year. Users run into this bug every four years...

```
>>> month_day = "02/29"
>>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  # No leap year bug.
datetime.datetime(1984, 2, 29, 0, 0)
```

Deprecated since version 3.13, will be removed in version 3.15: *strptime()* calls using a format string containing a day of month without a year now emit a *DeprecationWarning*. In 3.15 or later we may change this into an error or change the default year to a leap year. See gh-70647.

🄵解

8.2 zoneinfo --- IANA 時區支援

在 3.9 版被加入.

原始碼： Lib/zoneinfo

---

zoneinfo 模組提供了一個具體的時區實作，以支援 **PEP 615** 中最初指定的 IANA 時區資料庫。預設情🄵下，如果可用，*zoneinfo* 會使用系統的時區資料；如果🄵有可用的系統時區資料，該函式庫將轉而使用 PyPI 上可用的第一方 tzdata 套件。

也參考

***datetime* 模組**
提供了*time* 與*datetime* 型🄵，這些型🄵是設計來與*ZoneInfo* 類🄵一起使用的。

tzdata 套件
由 CPython 核心開發者維護的第一方套件，用於透過 PyPI 提供時區資料。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

8.2.1 使用 ZoneInfo

ZoneInfo 是*datetime.tzinfo* 抽象基底類🄵的一個具體實作，旨在透過建構函式、*datetime.replace* 方法或*datetime.astimezone* 方法附加到 `tzinfo`：

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

以此方式建構的 Datetime 物件與 datetime 運算相容，🄵能處理日光節約時間轉🄵而無需進一步干預：

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

這些時區也支援 **PEP 495** 中引入的*fold* 屬性。在會引發時間模糊的時差轉🄵期間（例如日光節約時間到標準時間的轉🄵），當 `fold=0` 時，會使用轉🄵前的時差，而當 `fold=1` 時，會使用轉🄵後的時差，例如：

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

當從另一個時區轉🄵時，fold 將被設定🄵正確的值：

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # 在 PDT 轉🄵🄵 PST 之前
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # 在 PDT 轉🄵🄵 PST 之後
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 資料來源

`zoneinfo` 模組不直接提供時區資料，而是從系統時區資料庫或可用的第一方 PyPI 套件 tzdata 中提取時區資訊。某些系統，特🄵是 Windows 系統，🄵有可用的 IANA 資料庫，因此對於需要時區資料且目標🄵跨平台相容性的專案，建議宣告對 tzdata 的依賴。如果系統資料和 tzdata 都不可用，所有對 *ZoneInfo* 的呼叫都將引發 *ZoneInfoNotFoundError* 例外。

設定資料來源

當呼叫 `ZoneInfo(key)` 時，建構函式會先在 *TZPATH* 中指定的目🄵中搜尋符合 `key` 的檔案，如果失敗，則在 tzdata 套件中尋找符合的項目。此行🄵可以透過三種方式設定：

1. 未另外指定時的預設 *TZPATH* 可以在編譯時期設定。
2. *TZPATH* 可以使用環境變數 來設定。
3. 在*runtime*，可以使用 *reset_tzpath()* 函式來操作搜尋路徑。

編譯時期設定

預設的 *TZPATH* 包含時區資料庫的幾個常見部署位置（Windows 除外，因🄵 Windows 上🄵有時區資料的「🄵所周知」位置）。在 POSIX 系統上，下游發行者和從原始碼建置 Python 且知道其系統時區資料部署位置的人，可以透過指定編譯時期選項 `TZPATH`（或更有可能的是使用 `configure` 旗標 `--with-tzpath`）來變更預設時區路徑，它應該是一個由 *os.pathsep* 分隔的字串。

在所有平台上，設定的值可以透過 *sysconfig.get_config_var()* 中的 `TZPATH` 鍵取得。

環境設定

在初始化 *TZPATH* 時（無論是在引入時，或是在呼叫不帶引數的 *reset_tzpath()* 時），如果環境變數 `PYTHONTZPATH` 存在，`zoneinfo` 模組將使用它來設定搜尋路徑。

PYTHONTZPATH

這是一個由 *os.pathsep* 分隔的字串，包含要使用的時區搜尋路徑。它必須只包含🄵對路徑，而非相對路徑。在 `PYTHONTZPATH` 中指定的相對路徑元件將不會被使用，但除此之外，指定相對路徑時的行🄵是實作定義的；CPython 將引發 *InvalidTZPathWarning*，但其他實作可以自由地🄵默忽略錯誤的元件或引發例外。

要讓系統忽略系統資料🄵改用 tzdata 套件，請設定 `PYTHONTZPATH=""`。

Runtime 設定

TZ 搜尋路徑也可以在 Runtime 使用 *reset_tzpath()* 函式進行設定。這通常不是一個建議的操作，🄵管在需要使用特定時區路徑（或需要停用對系統時區的存取）的測試函式中使用它是合理的。

8.2.3 ZoneInfo 類🄵

class zoneinfo.**ZoneInfo**(*key*)

一個具體的 *datetime.tzinfo* 子類🄵，代表由字串 key 指定的 IANA 時區。對主要建構函式的呼叫將總是回傳比較結果相同的物件；🄵句話🄵，除非透過 *ZoneInfo.clear_cache()* 使快取失效，否則對於所有 key 的值，以下斷言將永遠🄵真：

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

key 必須是相對、正規化的 POSIX 路徑形式，且不含上層參照。如果傳入不符合規範的鍵，建構函式將引發 *ValueError* 例外。

如果找不到符合 key 的檔案，建構函式將引發 *ZoneInfoNotFoundError* 例外。

ZoneInfo 類🄵有兩個備用建構函式：

classmethod ZoneInfo.**from_file**(*file_obj*, */*, *key=None*)

從回傳位元組的類檔案物件（例如以二進位模式開🄵的檔案或一個 *io.BytesIO* 物件）建構一個 ZoneInfo 物件。與主要建構函式不同，這個建構函式總是會建構一個新物件。

key 參數設定了時區的名稱，用於 `__str__()` 和 `__repr__()`。

透過此建構函式建立的物件無法被封裝（參🄵*pickling*）。

classmethod ZoneInfo.**no_cache**(*key*)

一個繞過建構函式快取的備用建構函式。它與主要建構函式相同，但每次呼叫都會回傳一個新物件。這對於測試或示範目的最可能有用，但它也可以用來建立一個具有不同快取失效策略的系統。

透過此建構函式建立的物件，在拆封時也會繞過去序列化過程的快取。

> **警示**
>
> 使用此建構函式可能會以意想不到的方式改變你的 datetime 物件的語意，只有在你確定需要時才使用它。

以下類🄵方法也可用：

classmethod ZoneInfo.**clear_cache**(**, only_keys=None*)

一個用於使 ZoneInfo 類🄵上的快取失效的方法。如果🄵有傳入引數，所有快取都會失效，且下一次對每個鍵的主要建構函式呼叫都將回傳一個新的實例。

如果將一個包含鍵名的可🄵代物件傳遞給 only_keys 參數，則只有指定的鍵會從快取中移除。傳遞給 only_keys 但在快取中找不到的鍵會被忽略。

> **警告**
>
> 呼叫此函式可能會以意想不到的方式改變使用 ZoneInfo 的 datetime 物件的語意；這會修改模組狀態，因此可能🄵生廣泛的影響。只有在你確定需要時才使用它。

該類🄵有一個屬性：

ZoneInfo.**key**

這是一個唯讀屬性，它回傳傳遞給建構函式的 key 值，該值應🄵 IANA 時區資料庫中的查詢鍵（例如 America/New_York、Europe/Paris 或 Asia/Tokyo）。

對於從檔案建構且未指定 key 參數的時區，此屬性將被設定🄵 None。

> **備🄵**
>
> 雖然將這些值暴露給終端使用者是一種相當普遍的做法，但這些值被設計🄵表示相關時區的主鍵，而不一定是面向使用者的元素。像 CLDR（Unicode 通用地區資料儲存庫）這樣的專案可以用來從這些鍵中取得更友善的使用者字串。

字串表示

在*ZoneInfo* 物件上呼叫*str* 時回傳的字串表示，預設使用*ZoneInfo.key* 屬性（參🄵屬性文件中的使用🄵明）：

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

對於從檔案建構且未指定 key 參數的物件，str 會轉而呼叫*repr()*。ZoneInfo 的 repr 是實作定義的，且不保證在不同版本之間保持穩定，但保證它不會是一個有效的 ZoneInfo 鍵。

封裝（Pickle）序列化

ZoneInfo 物件不是序列化所有轉🄵資料，而是按鍵進行序列化，而從檔案建構的 ZoneInfo 物件（即使是那些指定了 key 值的物件）也無法被封裝。

ZoneInfo 檔案的行🄵取🄵於它的建構方式：

1. ZoneInfo(key)：當使用主要建構函式建構時，ZoneInfo 物件會按鍵序列化，而在去序列化時，去序列化過程會使用主要建構函式，因此預期這些物件會與指向相同時區的其他參照是同一個物件。例如，如果 europe_berlin_pkl 是一個包含從 ZoneInfo("Europe/Berlin") 建構🄵封裝的字串，會預期以下行🄵：

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. ZoneInfo.no_cache(key)：當從繞過快取的建構函式建構時，ZoneInfo 物件也會按鍵序列化，但在去序列化時，去序列化過程會使用繞過快取的建構函式。如果 europe_berlin_pkl_nc 是一個包含從 ZoneInfo.no_cache("Europe/Berlin") 建構🄵封裝的字串，會預期以下行🄵：

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. ZoneInfo.from_file(file_obj, /, key=None)：當從檔案建構時，ZoneInfo 物件在封裝時會引發例外。如果終端使用者想要封裝一個從檔案建構的 ZoneInfo，建議他們使用包裝器型🄵或自訂序列化函式：可以按鍵序列化，或儲存檔案物件的🄵容🄵將其序列化。

這種序列化方法要求所需鍵的時區資料在序列化和去序列化兩端都可用，這與對類🄵和函式的參照預期存在於序列化和去序列化環境中的方式相似。這也意味著，當在具有不同版本時區資料的環境中拆封一個封裝的 ZoneInfo 時，不保證結果的一致性。

8.2.4 函式

zoneinfo.**available_timezones**()

取得一個集合，其中包含時區路徑上任何地方可用的所有 IANA 時區的有效鍵。這會在每次呼叫函式時重新計算。

此函式僅包含規範的時區名稱，不包含「特殊」時區，例如 posix/ 和 right/ 目🄵下的時區，或 posixrules 時區。

警示

此函式可能會開🄵大量檔案，因🄵確定時區路徑上的檔案是否🄵有效時區的最佳方法是讀取開頭的「魔術字串」。

備🄵

這些值🄵非設計來暴露給終端使用者；對於面向使用者的元素，應用程式應使用像 CLDR（Unicode 通用地區資料儲存庫）這樣的工具來取得更友善的使用者字串。另請參🄵關於 *ZoneInfo.key* 的警示🄵明。

zoneinfo.**reset_tzpath**(*to=None*)

🄵模組設定或重設時區搜尋路徑（*TZPATH*）。當不帶引數呼叫時，*TZPATH* 會被設定🄵預設值。

呼叫 reset_tzpath 不會使 *ZoneInfo* 快取失效，因此對主要 ZoneInfo 建構函式的呼叫只會在快取未命中時才使用新的 TZPATH。

to 參數必須是一個由字串或 *os.PathLike* 組成的*序列*，而不是一個字串，其中所有元素都必須是🄵對路徑。如果傳入的不是🄵對路徑，將會引發 *ValueError* 例外。

8.2.5 全域變數

zoneinfo.**TZPATH**

一個表示時區搜尋路徑的唯讀序列——當從一個鍵建構 ZoneInfo 時，該鍵會與 TZPATH 中的每個條目結合，🄵使用找到的第一個檔案。

TZPATH 只能包含🄵對路徑，🄵不能包含相對路徑，無論如何設定。

zoneinfo.TZPATH 指向的物件可能會因應對 *reset_tzpath()* 的呼叫而改變，因此建議使用 zoneinfo.TZPATH，而不是從 zoneinfo 引入 TZPATH 或將一個長生命🄵期的變數賦值給 zoneinfo.TZPATH。

有關設定時區搜尋路徑的更多資訊，請參🄵*設定資料來源*。

8.2.6 例外與警告

exception zoneinfo.**ZoneInfoNotFoundError**

當建構 *ZoneInfo* 物件因在系統上找不到指定的鍵而失敗時引發。這是 *KeyError* 的一個子類🄵。

exception zoneinfo.**InvalidTZPathWarning**

當 *PYTHONTZPATH* 包含將被過濾掉的無效元件（例如相對路徑）時引發。

8.3 calendar --- 日🄵相關函式

原始碼： Lib/calendar.py

這個模組讓你可以像 Unix 的 **cal** 程式一樣輸出日𝔽，𝔽額外提供有用的日𝔽相關函式。這些日𝔽預設把𝔽一當作一𝔽的第一天，而𝔽日當作最後一天（歐洲的慣例）。可以使用*setfirstweekday()* 設定一𝔽的第一天𝔽𝔽日 (6) 或一𝔽的其它任一天，其中指定日期的參數是整數。相關功能參考*datetime* 和*time* 模組。

這個模組定義的函式和類𝔽使用理想化的日𝔽，也就是公𝔽 (Gregorian calendar) 無限往前後兩個方向延伸。這符合 Dershowitz 和 Reingold 在「Calendrical Calculations」這本書定義的「逆推公𝔽」(proleptic Gregorian)，是做所有計算的基礎日𝔽。0 及負數年份的解讀跟 ISO 8601 標准𝔽規定的一樣，0 年是公元前 1 年，-1 年是公元前 2 年依此類推。

class calendar.**Calendar**(*firstweekday=0*)

建立*Calendar* 物件。*firstweekday* 是一個指定一𝔽第一天的整數，*MONDAY* 是 0（預設值），*SUNDAY* 是 6。

Calendar 物件提供一些方法來𝔽日𝔽資料的格式化做准備。這個類𝔽本身不做任何格式化，這是子類𝔽的工作。

Calendar 實例有以下方法與屬性：

firstweekday

The first weekday as an integer (0--6).

This property can also be set and read using *setfirstweekday()* and *getfirstweekday()* respectively.

getfirstweekday()

Return an *int* for the current first weekday (0--6).

Identical to reading the *firstweekday* property.

setfirstweekday(*firstweekday*)

Set the first weekday to *firstweekday*, passed as an *int* (0--6)

Identical to setting the *firstweekday* property.

iterweekdays()

回傳一個以數字代表一𝔽的每一天的𝔽代器 (iterator)。𝔽代器的第一個值和*firstweekday* 屬性的值一樣。

itermonthdates(*year*, *month*)

回傳一個在 *year* 年 *month* (1--12) 月的𝔽代器。這個𝔽代器會回傳該月的所有日期（*datetime.date* 物件）以及在該月之前及之後用來組成完整一𝔽的日期。

itermonthdays(*year*, *month*)

類似*itermonthdates()*，回傳一個在 *year* 年 *month* 月的𝔽代器，但不受限於*datetime.date* 的範圍。回傳的日期單純是該月當日的數字，對於該月之外的日期數字會是 0。

itermonthdays2(*year*, *month*)

類似*itermonthdates()*，回傳一個在 *year* 年 *month* 月的𝔽代器，但不受限於*datetime.date* 的範圍。回傳的日期是一個由該月當日的數字及代表𝔽幾的數字組成的元組。

itermonthdays3(*year*, *month*)

類似*itermonthdates()*，回傳一個在 *year* 年 *month* 月的𝔽代器，但不受限於*datetime.date* 的範圍。回傳的日期是一個由年、月、日的數字組成的元組。

在 3.7 版被加入.

itermonthdays4(*year*, *month*)

類似*itermonthdates()*，回傳一個在 *year* 年 *month* 月的𝔽代器，但不受限於*datetime.date* 的範圍。回傳的日期是一個由年、月、日及代表𝔽幾的數字組成的元組。

在 3.7 版被加入.

monthdatescalendar(*year*, *month*)

回傳一個在 *year* 年 *month* 月每一🄵組成的串列。每一🄵是一個串列，包含七個*datetime.date* 物件。

monthdays2calendar(*year*, *month*)

回傳一個在 *year* 年 *month* 月每一🄵組成的串列。每一🄵是一個串列，包含七個該月當日的數字及代表🄵幾的數字組成的元組。

monthdayscalendar(*year*, *month*)

回傳一個在 *year* 年 *month* 月每一🄵組成的串列。每一🄵是一個串列，包含七個該月當日的數字。

yeardatescalendar(*year*, *width=3*)

回傳用來格式化的指定年份的資料。回傳值是月份列的串列，每個月份列最多由 *width* 個月份組成（預設🄵 3）。每個月份包含四到六🄵，每一🄵包含一到七天，每一天則是一個*datetime.date* 物件。

yeardays2calendar(*year*, *width=3*)

回傳用來格式化的指定年份的資料（類似*yeardatescalendar()*）。每一天是一個該月當日的數字及代表🄵幾的數字組成的元組，該月外的日期的該月當日數字🄵 0。

yeardayscalendar(*year*, *width=3*)

回傳用來格式化的指定年份的資料（類似*yeardatescalendar()*）。每一天是一個該月當日的數字，該月外的日期的該月當日數字🄵 0。

class calendar.**TextCalendar**(*firstweekday=0*)

這個類🄵用來🄵生純文字的日🄵。

TextCalendar 實例有以下方法：

formatday(*theday*, *weekday*, *width*)

Return a string representing a single day formatted with the given *width*. If *theday* is `0`, return a string of spaces of the specified width, representing an empty day. The *weekday* parameter is unused.

formatweek(*theweek*, *w=0*)

Return a single week in a string with no newline. If *w* is provided, it specifies the width of the date columns, which are centered. Depends on the first weekday as specified in the constructor or set by the *setfirstweekday()* method.

formatweekday(*weekday*, *width*)

Return a string representing the name of a single weekday formatted to the specified *width*. The *weekday* parameter is an integer representing the day of the week, where `0` is Monday and `6` is Sunday.

formatweekheader(*width*)

Return a string containing the header row of weekday names, formatted with the given *width* for each column. The names depend on the locale settings and are padded to the specified width.

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

以多行字串的形式回傳一個月份的日🄵。如果給定 *w*，它會指定置中的日期欄的寬度。如果給定 *l*，它會指定每一🄵使用的行數。這個日🄵會依據在建構函式中指定或者透過*setfirstweekday()* 方法設定的一🄵的第一天來輸出。

formatmonthname(*theyear*, *themonth*, *width=0*, *withyear=True*)

Return a string representing the month's name centered within the specified *width*. If *withyear* is `True`, include the year in the output. The *theyear* and *themonth* parameters specify the year and month for the name to be formatted respectively.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

印出一個月份的日🄵，🄵容和*formatmonth()* 回傳的一樣。

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

以多行字串的形式回傳有 *m* 欄的一整年的日🄵。可選的參數 *w*、*l* 及 *c* 分🄵是日期欄寬度、每🄵行數及月份欄中間的空白數。這個日🄵會依據在建構函式中指定或者透過*setfirstweekday()* 方法設定的一🄵的第一天來輸出。最早可以🄵生日🄵的年份會依據平台而不同。

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

印出一整年的日🄵，🄵容和*formatyear()* 回傳的一樣。

class calendar.**HTMLCalendar**(*firstweekday=0*)

這個類🄵用來🄵生 HTML 日🄵。

HTMLCalendar 實例有以下方法：

formatmonth(*theyear*, *themonth*, *withyear=True*)

以 HTML 表格的形式回傳一個月份的日🄵。如果 *withyear* 是 true 則標題會包含年份，否則只會有月份名稱。

formatyear(*theyear*, *width=3*)

以 HTML 表格的形式回傳一整年的日🄵。*width*（預設🄵 3）指定一列有幾個月。

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

以完整 HTML 頁面的形式回傳一整年的日🄵。*width*（預設🄵 3）指定一列有幾個月。*css* 是要使用的 CSS (cascading style sheet) 名稱，可以給*None* 表示不使用任何 CSS。*encoding* 指定輸出使用的編碼（預設使用系統預設編碼）。

formatmonthname(*theyear*, *themonth*, *withyear=True*)

以 HTML 表列的形式回傳一個月份的名稱。如果 *withyear* 是 true 則該列會包含年份，否則只會有月份名稱。

HTMLCalendar 可以覆寫以下屬性來客🄵日🄵所使用的 CSS 類🄵：

cssclasses

對應一🄵每一天 CSS 類🄵的串列。預設的串列🄵容🄵：

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

可以針對每一天增加更多樣式：

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

注意這個串列的長度必須是七個項目。

cssclass_noday

跟當月🄵同一🄵且屬於前一個或下一個月份的日期使用的 CSS 類🄵。

在 3.7 版被加入.

cssclasses_weekday_head

在標題列中一🄵每一天名稱的 CSS 類🄵的串列。預設🄵容和*cssclasses* 相同。

在 3.7 版被加入.

cssclass_month_head

月份標題的 CSS 類🄵（由*formatmonthname()* 所使用），預設值是 "month"。

在 3.7 版被加入.

cssclass_month

整個月份表格的 CSS 類🄵（由*formatmonth()* 所使用），預設值是 "month"。

在 3.7 版被加入.

cssclass_year

整年表格的 CSS 類🄵（由 *formatyear()* 所使用），預設值是 "year"。

在 3.7 版被加入.

cssclass_year_head

整年表格標題的 CSS 類🄵（由 *formatyear()* 所使用），預設值是 "year"。

在 3.7 版被加入.

注意雖然上面提到的 CSS 屬性名稱是單數（例如 cssclass_month、cssclass_noday），你可以使用多個以空格隔開的 CSS 類🄵取代單一 CSS 類🄵，例如：

```
"text-bold text-red"
```

以下是客🄵化 HTMLCalendar 的範例：

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

TextCalendar 的子類🄵，可以在建構函式傳入語系名稱，它會回傳指定語系的月份及一🄵每一天的名稱。

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

HTMLCalendar 的子類🄵，可以在建構函式傳入語系名稱，它會回傳指定語系的月份及一🄵每一天的名稱。

備🄵

這兩個類🄵的建構函式、formatweekday() 及 formatmonthname() 方法會把 LC_TIME 語系暫時改成給定的 *locale*。因🄵目前的語系是屬於整個行程 (process-wide) 的設定，它們不是執行緒安全的。

這個模組提供以下函式給單純的文字日🄵使用。

calendar.**setfirstweekday**(*weekday*)

設定一🄵的第一天（0 是🄵一、6 是🄵日）。提供*MONDAY*、*TUESDAY*、*WEDNESDAY*、*THURSDAY*、*FRIDAY*、*SATURDAY* 及*SUNDAY* 可以方便設定。例如設定一🄵的第一天🄵🄵日：

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

回傳目前設定的一🄵的第一天。

calendar.**isleap**(*year*)

如果 *year* 是閏年回傳*True*，否則回傳*False*。

calendar.**leapdays**(*y1*, *y2*)

回傳從 *y1* 到 *y2*（不包含）間有幾個閏年，其中 *y1* 和 *y2* 是年份。

這個函式也適用在跨越世紀的時間範圍。

calendar.**weekday**(*year*, *month*, *day*)

回傳 *year* 年 (1970--...) *month* 月 (1--12) *day* 日 (1--31) 是🄵幾（0 是星期一）。

calendar.**weekheader**(*n*)

回傳包含一🄵每一天的名稱縮寫的標題。*n* 指定每一天的字元寬度。

calendar.**monthrange**(*year*, *month*)

回傳指定 *year* 年 *month* 月該月第一天代表🄵幾的數字及該月有多少天。

calendar.**monthcalendar**(*year*, *month*)

回傳代表一個月份日🄵的矩陣。每一列🄵一🄵；該月以外的日期以 0 表示。每一🄵以🄵一開始，除非有使用*setfirstweekday()* 改變設定。

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)

印出一個月份的日🄵，跟*month()* 回傳的🄵容一樣。

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)

以多行字串的形式回傳一個月的日🄵，使用*TextCalendar* 類🄵的*formatmonth()*。

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

印出一整年的日🄵，跟*calendar()* 回傳的🄵容一樣。

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

以多行字串回傳三欄形式的一整年日🄵，使用*TextCalendar* 類🄵的*formatyear()*。

calendar.**timegm**(*tuple*)

一個跟日🄵無關但方便的函式，它接受一個像*time* 模組🄵的*gmtime()* 函式回傳的元組，🄵回傳對應的 Unix 時間戳，假設從 1970 開始及 POSIX 編碼。事實上，*time.gmtime()* 和*timegm()* 是彼此相反的。

calendar 模組匯出以下資料屬性：

calendar.**day_name**

以目前語系來表示的一🄵每一天名稱的序列，其中🄵一是第 0 天。

```
>>> import calendar
>>> list(calendar.day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

calendar.**day_abbr**

以目前語系來表示的一🄵每一天縮寫名稱的序列，其中 Mon 是第 0 天。

```
>>> import calendar
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

calendar.**MONDAY**

calendar.**TUESDAY**

calendar.**WEDNESDAY**

calendar.**THURSDAY**

calendar.**FRIDAY**

calendar.**SATURDAY**

calendar.**SUNDAY**

一🄵每一天的🄵名，其中 `MONDAY` 是 `0` 而 `SUNDAY` 是 `6`。

在 3.12 版被加入.

class calendar.**Day**

將一🄵中的幾天定義🄵整數常數的列舉。此列舉的成員將作🄵*MONDAY* 到*SUNDAY* 匯出到模組作用域。

在 3.12 版被加入.

calendar.**month_name**

以目前語系來表示的一年每個月份名稱的序列。它按照一般慣例以數字 1 代表一月，因此它的長度🄵 13，而 month_name[0] 是空字串。

```
>>> import calendar
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
↪'September', 'October', 'November', 'December']
```

calendar.**month_abbr**

以目前語系來表示的一年每個月份縮寫名稱的序列。它按照一般慣例以數字 1 代表一月，因此它的長度🄵 13，而 month_abbr[0] 是空字串。

```
>>> import calendar
>>> list(calendar.month_abbr)
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

calendar.**JANUARY**

calendar.**FEBRUARY**

calendar.**MARCH**

calendar.**APRIL**

calendar.**MAY**

calendar.**JUNE**

calendar.**JULY**

calendar.**AUGUST**

calendar.**SEPTEMBER**

calendar.**OCTOBER**

calendar.**NOVEMBER**

calendar.**DECEMBER**

一年🄵每個月的🄴名，其中 JANUARY 是 1 而 DECEMBER 是 12。

在 3.12 版被加入.

class calendar.**Month**

將一年中的月份定義🄴整數常數的列舉。此列舉的成員將作🄵*JANUARY* 到*DECEMBER* 匯出到模組作用域。

在 3.12 版被加入.

calendar 模組定義了以下例外：

exception calendar.**IllegalMonthError**(*month*)

ValueError 的子類🄴，當給定的月份數字超出 1-12 範圍（含）時引發。

month

無效的月份號。

exception calendar.**IllegalWeekdayError**(*weekday*)

ValueError 的子類🄴，當給定的🄵幾的數字超出 0-6（含）範圍時引發。

weekday

無效的🄵幾編號。

也參考

***datetime* 模組**

日期與時間的物件導向介面，和*time* 模組有相似的功能。

time **模組**
底層的時間相關函式。

8.3.1 命令列用法

在 2.5 版被加入.

calendar 模組可以作🄵🄵本從命令列執行，🄵以互動方式列印日🄵。

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t {text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

例如，要列印 2000 年的日🄵:

```
$ python -m calendar 2000
                                  2000

      January                   February                  March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6            1  2  3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9 10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16 17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23 24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30 31
31

       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7               1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                    August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                  1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                  1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

接受以下選項:

--help, -h
顯示幫助訊息🄵退出。

--locale LOCALE, **-L** LOCALE
用於月份和🄵幾名稱的語系。預設🄵英語。

--encoding ENCODING, **-e** ENCODING

用於輸出的編碼。如有設定*--locale*則必須給定*--encoding*。

--type {text,html}, **-t** {text,html}

將日🄵以文字或 HTML 文件的形式印出到終端機。

--first-weekday FIRST_WEEKDAY, **-f** FIRST_WEEKDAY

一🄵起始的日子。必須是 0（🄵一）到 6（🄵日）之間的數字。預設🄵 0。

在 3.13 版被加入.

year

印出日🄵的年份。預設🄵目前年份。

month

要列印日🄵的指定*year*的月份。必須是 1 到 12 之間的數字，🄵且只能在文字模式下使用。預設列印全年日🄵。

文字模式選項：

--width WIDTH, **-w** WIDTH

終端機行中日期行的寬度。日期印出在行的中央。任何小於 2 的值都會被忽略。預設🄵 2。

--lines LINES, **-l** LINES

終端機列中每🄵的列數。日期印出時頂部會對齊。任何小於 1 的值都會被忽略。預設🄵 1。

--spacing SPACING, **-s** SPACING

行中月份之間的間距。任何小於 2 的值都會被忽略。預設🄵 6。

--months MONTHS, **-m** MONTHS

每列印出的月份數量。預設🄵 3。

在 3.14 版的變更: By default, today's date is highlighted in color and can be controlled using environment variables.

HTML 模式選項：

--css CSS, **-c** CSS

用於日🄵的 CSS 樣式表路徑。這必須是相對於🄵生之 HTML 的，或者🄵🄵對的 HTTP 或 `file:///` URL。

8.4 `collections` --- 容器資料型態

原始碼： Lib/collections/__init__.py

這個模組實作了一些特🄵的容器資料型態，用來替代 Python 一般🄵建的容器，例如*dict*（字典）、*list*（串列）、*set*（集合）和*tuple*（元組）。

namedtuple()	用來建立具名欄位的 tuple 子類🄵的工廠函式
deque	一個類似 list 的容器，可以快速的在頭尾加入 (append) 元素與移除 (pop) 元素
ChainMap	一個類似 dict 的類🄵，用來🄵多個對映 (mapping) 建立單一的視圖 (view)
Counter	dict 的子類🄵，用來計算可雜🄵物件的數量
OrderedDict	dict 的子類🄵，會記🄵物件被加入的順序
defaultdict	dict 的子類🄵，當值不存在 dict 中時會呼叫一個提供預設值的工廠函式
UserDict	dict 物件的包裝器 (wrapper)，簡化了 dict 的子類🄵化過程
UserList	list 物件的包裝器，簡化了 list 的子類🄵化過程
UserString	字串物件的包裝器，簡化了字串的子類🄵化過程

8.4.1 ChainMap 物件

在 3.3 版被加入.

ChainMap（對映鏈結）類🄵的目的是快速將數個對映連結在一起，讓它們可以被當作一個單元來處理。它通常會比建立一個新的字典🄵多次呼叫*update()* 來得更快。

這個類🄵可用於模擬巢狀作用域 (nested scopes)，且在模板化 (templating) 時能派上用場。

class collections.**ChainMap**(**maps*)

一個*ChainMap* 將多個 dict 或其他對映組合在一起，建立一個獨立、可更新的視圖。如果🄵有指定 *maps*，預設會提供一個空字典讓每個新鏈結都至少有一個對映。

底層的對映儲存於一個 list 中，這個 list 是公開的且可透過 *maps* 屬性存取或更新，🄵有其他狀態 (state)。

檢索 (lookup) 陸續查詢底層對映，直到鍵被找到，然而讀取、更新和🄵除就只會對第一個對映操作。

ChainMap 透過參照將底層對映合🄵，所以當一個底層對映被更新，這些改變也會反映到*ChainMap*。

所有常見的字典方法都有支援。此外，還有一個 *maps* 屬性 (attribute)、一個建立子上下文 (subcontext) 的方法、和一個能🄵存取除了第一個以外其他所有對映的特性 (property)：

maps

一個可被更新的對映列表，這個列表是按照被搜尋的順序來排列，在 ChainMap 中它是唯一被儲存的狀態，可被修改來變🄵搜尋順序。回傳的列表都至少包含一個對映。

new_child(*m=None*, ***kwargs*)

回傳包含一個新對映的*ChainMap*，新對映後面接著目前實例的所有現存對映。如果有給定 `m`，`m` 會成🄵那個最前面的新對映；若🄵有指定，則會加上一個空 dict，如此一來呼叫 `d.new_child()` 就等同於 `ChainMap({}, *d.maps)`。這個方法用於建立子上下文，而保持父對映的不變。

在 3.4 版的變更: 加入可選參數 `m`。

在 3.10 版的變更: 增加了對關鍵字引數的支援。

parents

回傳一個包含除了第一個以外所有其他對映的新*ChainMap* 的特性，可用於需要跳過第一個對映的搜尋。使用情境類似於在巢狀作用域當中使用 `nonlocal` 關鍵字，也可與🄵建函式*super()* 做類比。引用 `d.parents` 等同於 `ChainMap(*d.maps[1:])`。

注意，一個*ChainMap* 的🄵代順序是透過由後往前掃描對映而定：

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

這和呼叫*dict.update()* 結果的順序一樣是從最後一個對映開始：

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

在 3.9 版的變更: 支援 `|` 和 `|=` 運算子，詳見 **PEP 584**。

也參考

- Enthought CodeTools package 中的 MultiContext class 支援在鏈中選定任意對映寫入。
- Django 中用於模板的 Context class 是唯讀的對映鏈，也具有加入 (push) 和移除 (pop) 上下文的功能，與*new_child()* 方法和*parents* 特性類似。

- Nested Contexts recipe 提供了控制是否只對鏈中第一個或其他對映做寫入或其他操作的選項。
- 一個極度簡化、維讀版本的 Chainmap。

ChainMap 范例和用法

此章節提供了多種操作 ChainMap 的案例。

模擬 Python 🄴部檢索鏈結的例子：

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

讓使用者指定的命令列引數優先於環境變數、再優先於預設值的範例：

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

用*ChainMap*類🄴模擬巢狀上下文的範例模式：

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

*ChainMap*類🄴只對鏈結中第一個對映來做寫入或🄴除，但檢索則會掃過整個鏈結。但如果需要對更深層的鍵寫入或🄴除，透過定義一個子類🄴來實作也不困難：

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
```

(繼續下一頁)

(繼續上一頁)

```
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 `Counter` 物件

提供一個支援方便且快速計數的計數器工具，例如：

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

Counter 是*dict* 的子類🄵，用來計算可雜🄵物件的數量。它是將物件與其計數作🄵字典的鍵值對儲存的集合容器。計數可以是包含 0 與負數的任何整數值。*Counter* 類🄵類似其他程式語言中的 bags 或 multisets。

被計數的元素來自一個 *iterable* 或是被其他的 *mapping*（或 Counter）初始化：

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter 物件擁有一個字典的使用介面，除了遇到 Counter 中🄵有的值時會回傳計數 0 取代*KeyError* 這點不同：

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

將一個值的計數設🄵 0 🄵不會真的從 Counter 中🄵除這個元素，要使用 `del` 來將其🄵除：

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

在 3.1 版被加入.

在 3.7 版的變更: 作🄵*dict* 的子類🄵，*Counter* 繼承了記憶插入順序的功能。對 *Counter* 做數學運算後同樣保留順序性，其結果是依照各個元素在運算元左邊出現的時間先後、再按照運算元右邊出現的時間先後來排列。

除了字典原本就有的方法外，Counter 物件額外支援數個新方法：

elements()

回傳每個元素都重🄴出現計算次數的 iterator（🄴代器）物件，其中元素的回傳順序是依照各元素首次出現的時間先後。如果元素的出現次數小於 1，*elements()* 方法會忽略這些元素。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

回傳一個 list，包含出現最多次的 *n* 個元素及其出現次數，🄴按照出現次數排序。如果 *n* 被省略或者🄴 None，*most_common()* 會回傳所有 counter 中的元素。出現次數相同的元素會按照首次出現的時間先後來排列：

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

🄴去自一個 *iterable* 或另一個對映（或 Counter）中的計數元素，行🄴類似*dict.update()* 但是是🄴了🄴去計數而非取代其值。輸入和輸出都可以是 0 或是負數。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

在 3.2 版被加入.

total()

計算總計數值。

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

在 3.10 版被加入.

通常來🄴字典方法也可以用於*Counter* 物件，除了以下兩個作用方式與計數器不同。

fromkeys(*iterable*)

此類🄴方法🄴有被實作於*Counter* 物件中。

update([*iterable-or-mapping*])

加上自一個 *iterable* 計算出的計數或加上另一個 *mapping*（或 Counter）中的計數，行🄴類似*dict.update()* 但是是🄴了加上計數而非取代其值。另外，*iterable* 需要是一串將被計算個數元素的序列，而非元素🄴 `(key, value)` 形式的序列。

Counter 支援相等性、子集和超集關🄴的 rich comparison 運算子：`==`、`!=`、`<`、`<=`、`>`、`>=`。這些檢測會將不存在的元素之計數值當作零，因此 `Counter(a=1) == Counter(a=1, b=0)` 將回傳真值。

在 3.10 版的變更: 增加了 rich comparison 運算。

在 3.10 版的變更: 在相等性運算中，不存在的元素之計數值會被當作零。在此之前，`Counter(a=3)` 和 `Counter(a=3, b=0)` 被視🄴不同。

使用*Counter* 物件的常見使用模式：

```
c.total()                       # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
```

(繼續下一頁)

(繼續上一頁)

```
dict(c)                             # convert to a regular dictionary
c.items()                           # access the (elem, cnt) pairs
Counter(dict(list_of_pairs))        # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]           # n least common elements
+c                                  # remove zero and negative counts
```

🄵結合多個*Counter* 物件以🄵生 multiset（多重集合，擁有大於 0 計數元素的計數器），有提供了幾種數學操作。加法和🄵法是根據各個對應元素分🄵將 Counter 加上和🄵去計數，交集和聯集分🄵回傳各個元素最小和最大計數，相等性與包含性運算則會比較對應的計數。每一個操作都可以接受輸入帶有正負號的計數，但輸出的 Counter 則會將擁有小於或等於 0 計數的元素剔除。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                           # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                           # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                           # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                           # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                          # equality:  c[x] == d[x]
False
>>> c <= d                          # inclusion:  c[x] <= d[x]
False
```

加🄵法的一元運算子分🄵是加上空的 Counter 和從空 Counter 🄵去的簡寫。

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

在 3.3 版被加入: 開始支援加🄵一元運算子和 multiset 的原地 (in-place) 操作。

備🄵

Counter 主要是被設計來操作正整數以當作使用中的計數，但🄵了某些會用到計數之值🄵負數或🄵其他型🄵的案例中，Counter 也小心地被設計成不會預先排除這些特殊元素。🄵了輔助使用於上述案例，這一小節記🄵了最小範圍和型🄵限制。

- *Counter* 類🄵本身是字典的子類🄵，且不限制其鍵與值。值被用來表示計數，但實際上你可以儲存任何值。
- 使用*most_common()* 方法的唯一條件是其值要是可被排序的。
- 像是 `c[key] += 1` 的原地操作中，其值之型🄵只必須支援加🄵，所以分數、浮點數、十進位數與其負值都可以使用。同理，*update()* 和*subtract()* 也都允許 0 或負值🄵輸入或輸出。
- Multiset 相關方法只🄵了處理正值而設計，其輸入允許是 0 或負值但只有正值會被輸出。🄵無型🄵限制，但其值的型🄵須支援加、🄵及比較運算。
- *elements()* 方法需要其計數🄵正值，如🄵 0 或負值則忽略。

也參考

- Smalltalk 中的 Bag class。
- 維基百科上的多重集合條目。

- C++ multisets 教學與範例。
- Multiset 的數學運算及其使用時機，參考 *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*。
- 若要根據給定的元素集合來列舉出所有不重🄵且擁有指定元素數量的 multiset，請見 *itertools.combinations_with_replacement()*：

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.4.3 deque 物件

class collections.**deque**([*iterable*[, *maxlen*]])

回傳一個新的 deque（雙端🄵列）物件，將 *iterable* 中的資料由左至右（使用 *append()*）加入來做初始化。如果 *iterable* 🄵未給定，回傳的則是一個空的 deque。

Deque（發音🄵"deck"，🄵"double-ended queue" 的簡稱）🄵 stack 和 queue 的一般化。deque 支援執行緒安全 (thread-safe)，且能🄵有效率地節省記憶體在頭和尾加入和移除元素，兩個方向的表現都大致🄵 *O*(1) 🄵雜度。

雖然 *list* 物件也支援類似操作，但 list 優化了長度固定時的操作，而會改變底層資料的長度及位置的 `pop(0)` 和 `insert(0, v)` 操作，記憶體移動則🄵 *O*(*n*) 🄵雜度。

如果 *maxlen* 🄵有給定或者🄵 `None`，deque 可以增長到任意長度；但若有給定的話，deque 的最大長度就會被限制。一個被限制長度的 deque 一但滿了，若在一端加入數個新元素，則同時會在另一端移除相同數量的元素。限定長度的 deque 提供了和 Unix `tail` filter 類似的功能，可用於追🄵使用者在意的那些最新執行事項或數據源。

Deque 物件支援以下方法：

append(*x*)

將 *x* 自 deque 的右側加入。

appendleft(*x*)

將 *x* 自 deque 的左側加入。

clear()

將所有元素從 deque 中移除，使其長度🄵 0。

copy()

建立一個 deque 的淺🄵🄵 (shallow copy)。

在 3.5 版被加入.

count(*x*)

計算 deque 🄵元素🄵 *x* 的個數。

在 3.2 版被加入.

extend(*iterable*)

將 iterable 引數加入 deque 的右側。

extendleft(*iterable*)

將 iterable 引數加入 deque 的左側。要注意的是，加入後的元素順序和 iterable 參數是相反的。

index(*x*[, *start*[, *stop*]])

回傳 deque 中 *x* 的位置（或在索引 *start* 之後、索引 *stop* 之前的位置）。回傳第一個匹配的位置，如果🄵找到就引發 *ValueError*。

在 3.5 版被加入.

insert(*i*, *x*)

在 deque 位置 *i* 中插入 *x*。

如果此插入操作導致 deque 超過其長度上限 *maxlen* 的話，會引發*IndexError* 例外。

在 3.5 版被加入.

pop()

移除🄵回傳 deque 的最右側元素，若本來就🄵有任何元素，則會引發*IndexError*。

popleft()

移除🄵回傳 deque 的最左側元素，若本來就🄵有任何元素，則會引發*IndexError*。

remove(*value*)

移除第一個出現的 *value*，如果🄵找到的話就引發一個*ValueError*。

reverse()

將 deque 中的元素原地 (in-place) 倒序排列🄵回傳 `None`。

在 3.2 版被加入.

rotate(*n=1*)

將 deque 向右輪轉 *n* 步。若 *n* 🄵負值則向左輪轉。

當 deque 不是空的，向右輪轉一步和 `d.appendleft(d.pop())` 有相同意義，而向左輪轉亦等價於 `d.append(d.popleft())`。

Deque 物件也提供了一個唯讀屬性:

maxlen

Deque 的最大長度，如果不限制長度的話則🄵 `None`。

在 3.1 版被加入.

除了以上使用方式，deque 亦支援了🄵代、pickle、`len(d)`、`reversed(d)`、`copy.copy(d)`、`copy.deepcopy(d)`、用 `in` 運算子來作隸屬資格檢測以及像是 `d[0]` 的標號引用來取得第一個元素。在兩端做索引存取的🄵雜度🄵 *O*(1) 但越靠近中間則🄵慢至 *O*(*n*)。若想要隨機而快速的存取，使用 list 會較🄵合適。

自從 3.5 版本起，deque 開始支援 `__add__()`、`__mul__()` 和 `__imul__()`。

範例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
```

(繼續下一頁)

(繼續上一頁)

```
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque 用法

這一章節提供了多種操作 deque 的案例。

被限制長度的 deque 功能類似 Unix 中的 `tail` filter：

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

另一用法是透過從右邊加入、從左邊移除來維護最近加入元素的 list：

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

一個輪詢調度器可以透過在*deque* 中放入 iterator 來實現，值自目前 iterator 的位置 0 取出，如果 iterator 已經消耗完畢就用*popleft()* 將其從🄵列中移除，否則利用*rotate()* 來將其移至🄵列尾端：

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
```

(繼續下一頁)

(繼續上一頁)

```
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

rotate() 提供了可以用來實作*deque* 切片和🄴除的方法。舉例來🄴，用純 Python 實作 `del d[n]` 需要用 `rotate()` 來定位要被移除的元素：

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

要實現*deque* 切片，可使用近似以下方法：使用*rotate()* 來將目標元素移動到 deque 最左側，用*popleft()* 移除舊元素🄴用*extend()* 加入新元素，最後再反向 rotate。在這個方法上做小小的更動就能簡單地實現 Forth 風格的 stack 操作，例如 `dup`、`drop`、`swap`、`over`、`pick`、`rot` 和 `roll`。

8.4.4 `defaultdict` 物件

class collections.**defaultdict**(*default_factory=None*, */*[, ...])

回傳一個新的類似字典的物件。*defaultdict* 是🄴建類🄵*dict* 的子類🄴。它覆蓋掉了一個方法🄴添加了一個可寫入的實例變數。其余功能與*dict* 相同，此文件不再🄴述。

第一個引數🄵*default_factory* 屬性提供了初始值，他被預設🄵 `None`，所有其他的引數（包括關鍵字引數）都會被傳遞給*dict* 的建構函式 (constructor)。

defaultdict 物件支援以下*dict* 所🄵有的方法：

__missing__(*key*)

如果*default_factory* 屬性🄵 `None`，呼叫此方法會引發一個附帶引數 *key* 的*KeyError* 例外。

如果*default_factory* 不🄵 `None`，它會不帶引數地被呼叫來🄵給定的 *key* 提供一個預設值，這個值和 *key* 被作🄵鍵值對來插入到字典中，🄵且被此方法所回傳。

如果呼叫*default_factory* 時發生例外，則該例外將會保持不變地向外傳遞。

在無法找到所要求的鍵時，此方法會被*dict* 類🄵的 `__getitem__()` 方法呼叫。無論此方法回傳了值還是引發了例外，都會被 `__getitem__()` 所傳遞。

注意，*__missing__()* 不會被 `__getitem__()` 以外的其他方法呼叫，這意味著*get()* 會像一般的 dict 那樣回傳 `None` 做🄴預設值，而非使用*default_factory*。

defaultdict 物件支援以下實例變數：

default_factory

此屬性🄵*__missing__()* 方法所使用。如果有引數被傳入建構函式，則此屬性會被初始化成第一個引數，如未提供引數則被初始化🄵 `None`。

在 3.9 版的變更: 新增合🄵 (|) 和更新 (|=) 運算子，請見 **PEP 584**。

`defaultdict` 范例

使用*list* 作🄵*default_factory* 可以很輕松地將鍵值對序列轉🄵🄵包含 list 之字典：

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

當每個鍵第一次被存取時，它還🄵有存在於對映中，所以會自動呼叫*default_factory* 方法來回傳一個空的*list* 以建立一個條目，*list.append()* 操作後續會再新增值到這個新的列表🄵。當再次存取該鍵時，就如普通字典般操作（回傳該鍵所對應到的 list），*list.append()* 也會新增另一個值到 list 中。和使用與其等價的*dict.setdefault()* 相比，這個技巧更加快速和簡單：

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

設定*default_factory* 🄵*int* 使得*defaultdict* 可被用於計數（類似其他語言中的 bag 或 multiset）：

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

當一個字母首次被存取時，它🄵不存在於對映中，則*default_factory* 函式會呼叫*int()* 來提供一個整數 0 作🄵預設值。後續的增加操作繼續對每個字母做計數。

函式*int()* 總是回傳 0，這是常數函式的特殊情🄵。一個更快、更有彈性的方法是使用 lambda 函式來提供任何常數值（不用一定要是 0）：

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

將*default_factory* 設🄵*set* 使*defaultdict* 可用於構建一個值🄵 set 的字典：

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 `namedtuple()` 擁有具名欄位之 tuple 的工廠函式

Named tuple（具名元組）賦予 tuple 中各個位置意義，使程式碼更有可讀性與自我文件性。它們可以用於任何普通 tuple 可使用的場合，賦予其透過名稱（而非位置索引）來存取欄位的能力。

`collections.`**`namedtuple`**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

回傳一個名🄵 *typename* 的新 tuple 子類🄵。這個新的子類🄵被用於建立類似 tuple 的物件，可以透過屬性名稱來存取欄位，它同時也是可索引 (indexable) 和可🄴代的 (iterable)。子類🄵實例同樣有文件字串 (docstring)（有類🄵名稱 *typename* 和欄位名 *field_names*）和一個好用的 `__repr__()` 方法，可將 tuple 🄵容以 `name=value` 格式列出。

field_names 是一個像 `['x', 'y']` 一樣的字串序列。*field_names* 也可以是一個用空白或逗號分隔各個欄位名稱的字串，比如 `'x y'` 或者 `'x, y'`。

除了底🄵開頭以外的其他任何有效 Python 識🄵字 (identifier) 都可以作🄵欄位名稱，有效識🄵字由字母、數字、底🄵所組成，但不能是數字或底🄵開頭，也不能是關鍵詞*keyword*，例如 *class*、*for*、*return*、*global*、*pass* 或 *raise*。

如果 *rename* 🄵真值，無效的欄位名稱會自動被位置名稱取代。比如 `['abc', 'def', 'ghi', 'abc']` 會被轉🄵成 `['abc', '_1', 'ghi', '_3']`，移除了關鍵字 `def` 和重🄵欄位名 `abc`。

defaults 可以🄵 `None` 或者是一個預設值的*iterable*。因🄵有預設值的欄位必須出現在那些🄵有預設值的欄位之後，*defaults* 是被應用在右側的引數。例如 fieldnames 🄵 `['x', 'y', 'z']` 且 defaults 🄵 `(1, 2)`，那🄵 `x` 就必須被給定一個引數，`y` 被預設🄵 `1`，`z` 則被預設🄵 `2`。

如果 *module* 值有被定義，named tuple 的 `__module__` 屬性就被設定🄵該值。

Named tuple 實例中🄵有字典，所以它們更加輕量，且和一般 tuple 相比🄵用更少記憶體。

要支援 pickle，應將 named tuple 類🄵賦值給一個符合 *typename* 的變數。

在 3.1 版的變更: 新增對於 *rename* 的支援。

在 3.6 版的變更: *verbose* 和 *rename* 參數成🄵僅限關鍵字引數。

在 3.6 版的變更: 新增 *module* 參數。

在 3.7 版的變更: 移除 *verbose* 參數和 `_source` 屬性。

在 3.7 版的變更: 新增 *defaults* 參數和*_field_defaults* 屬性。

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuple 在賦予欄位名稱於*csv* 或*sqlite3* 模組回傳之 tuple 時相當有用：

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

除了繼承自 tuple 的方法，named tuple 還支援三個額外的方法和兩個屬性。🄵了防止欄位名稱有衝突，方法和屬性的名稱以底🄵開頭。

classmethod somenamedtuple.**_make**(*iterable*)

從已存在的序列或可🄵代物件建立一個新實例的類🄵方法。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

回傳一個將欄位名稱對映至對應值的*dict*：

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

在 3.1 版的變更: 回傳一個*OrderedDict* 而非*dict*。

在 3.8 版的變更: 回傳一個常規*dict* 而非*OrderedDict*，自從 Python 3.7 開始，dict 已經保證有順序性，如果需要*OrderedDict* 所專屬的特性，推薦的解法是將結果專🄵成所需的類型: `OrderedDict(nt._asdict())`。

somenamedtuple.**_replace**(***kwargs*)

回傳一個新的 named tuple 實例，🄵將指定欄位替🄵🄵新的值:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.
↪now())
```

Named tuples are also supported by generic function *copy.replace()*.

在 3.13 版的變更: Raise *TypeError* instead of *ValueError* for invalid keyword arguments.

somenamedtuple.**_fields**

列出 tuple 欄位名稱的字串，用於自我檢查或是從現有 named tuple 建立一個新的 named tuple 型🄵。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

將欄位名稱對映至預設值的字典。

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

要取得這個名稱存於字串的欄位，要使用*getattr()* 函式:

```
>>> getattr(p, 'x')
11
```

（如 tut-unpacking-arguments 所述）將一個字典轉🄵成 named tuple，要使用 ** 雙星號運算子:

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

因🄵一個 named tuple 是一個常規的 Python 類🄵，我們可以很容易的透過子類🄵來新增或更改功能，以下是如何新增一個計算得到的欄位和固定寬度的輸出列印格式：

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面的子類🄵將 `__slots__` 設定🄵空 tuple，這樣一來就防止了字典實例被建立，因而保持了較低的記憶體用量。

子類🄵化無法用於增加新的、已被儲存的欄位，應當透過*_fields* 屬性以建立一個新的 named tuple 來實現：

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

透過直接賦值給 `__doc__`，可以自訂🄵明文件字串：

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

在 3.5 版的變更: 文件字串屬性變成可寫入。

也參考

- 關於🄵 named tuple 新增型🄵提示的方法，請參🄵*typing.NamedTuple*，它運用 `class` 關鍵字以提供了一個簡潔的表示法：

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- 關於以 dict 而非 tuple 🄵底層的可變命名空間，請參考*types.SimpleNamespace()*。
- *dataclasses* 模組提供了一個裝飾器和一些函式，用於自動將被生成的特殊方法新增到使用者定義的類🄵中。

8.4.6 `OrderedDict` 物件

Ordered dictionary（有序字典）就像常規字典一樣，但有一些與排序操作相關的額外功能，但由於🄵建的*dict* 類🄵現在已經有記憶插入順序的能力（Python 3.7 中確保了這種新行🄵），它們變得不那🄵重要了。

仍存在一些與*dict* 的不同之處：

- 常規的*dict* 被設計成非常擅長於對映相關操作，追🄵插入的順序🄵次要目標。
- *OrderedDict* 則被設計成擅長於重新排序相關的操作，空間效率、🄵代速度和更新操作的效能則🄵次要設計目標。

- *OrderedDict* 比起*dict* 更適合處理頻繁的重新排序操作，如在下方用法中所示，這讓它適合用於多種 LRU cache 的實作中。
- *OrderedDict* 之相等性運算會檢查順序是否相同。

  一個一般的*dict* 可以用 `p == q and all(k1 == k2 for k1, k2 in zip(p, q))` 來效仿有檢查順序的相等性運算。
- *OrderedDict* 類🄵的*popitem()* 方法有不同的函式簽名 (signature)，它接受傳入一個選擇性引數來指定要移除哪個元素。

  一個一般的*dict* 可以用 `d.popitem()` 來效仿 OrderedDict 的 `od.popitem(last=True)`，這保證會移除最右邊（最後一個）的元素。

  一個一般的*dict* 可以用 `(k := next(iter(d)), d.pop(k))` 來效仿 OrderedDict 的 `od.popitem(last=False)`，若最左邊（第一個）的元素存在，則將其回傳🄵移除。
- *OrderedDict* 有個*move_to_end()* 方法可有效率地將一個元素重新排列到任一端。

  一個一般的*dict* 可以用 `d[k] = d.pop(k)` 來效仿 OrderedDict 的 `od.move_to_end(k, last=True)`，這會將該鍵與其對應到的值移動至最右（最後面）的位置。

  一個一般的*dict* 🄵有和 OrderedDict 的 `od.move_to_end(k, last=False)` 等價的有效方式，這是將鍵與其對應到的值移動至最左（最前面）位置的方法。
- 在 Python 3.8 之前，*dict* 🄵🄵有 `__reversed__()` 方法。

class collections.**OrderedDict**([*items*])

回傳一個*dict* 子類🄵的實例，它具有專門用於重新排列字典順序的方法。

在 3.1 版被加入.

popitem(*last=True*)

Ordered dictionary 的*popitem()* 方法移除🄵回傳一個鍵值 (key, value) 對。如果 *last* 🄵真值，則按 LIFO 後進先出的順序回傳鍵值對，否則就按 FIFO (first-in, first-out) 先進先出的順序回傳鍵值對。

move_to_end(*key*, *last=True*)

將現有的 *key* 移動到 ordered dictionary 的任一端。如果 *last* 🄵真值（此🄵預設值）則將元素移至右端；如果 *last* 🄵假值則將元素移至左端。如果 *key* 不存在則會引發*KeyError*：

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

在 3.2 版被加入.

除了普通的對映方法，ordered dictionary 還支援了透過*reversed()* 來做倒序🄵代。

OrderedDict 物件之間的相等性運算是會檢查順序是否相同的，大致等價於 `list(od1.items())==list(od2.items())`。

OrderedDict 物件和其他*Mapping* 物件間的相等性運算則像普通字典一樣不考慮順序性，這使得*OrderedDict* 可於任何字典可使用的時機中被替🄵掉。

在 3.5 版的變更: *OrderedDict* 的項 (item)、鍵與值之視圖現在可透過*reversed()* 來倒序🄵代。

在 3.6 版的變更: 隨著 **PEP 468** 被核可，被傳入給*OrderedDict* 建構函式與其*update()* 方法的關鍵字引數之順序被保留了下來。

在 3.9 版的變更: 新增合🄵 (`|`) 和更新 (`|=`) 運算子，請見 **PEP 584**。

OrderedDict 范例與用法

建立一個能🄵記住鍵最後插入順序的 ordered dictionary 變體很簡單。如果新條目覆蓋了現有條目，則原本插入位置會被更改🄴移動至末端：

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

OrderedDict 在實現一個*functools.lru_cache()* 的變形版本時也非常有用：

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()   # { uncached_key : request_count }
        self.cache = OrderedDict()      # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests  # max number of uncached requests
        self.maxsize = maxsize          # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
```

(繼續下一頁)

(繼續上一頁)

```
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

8.4.7 UserDict 物件

UserDict 類🄵是作🄵 dict 物件的包裝器。因🄵已經可以直接自*dict* 建立子類🄵，這個類🄵的需求已部分被滿足，不過這個類🄵使用起來更方便，因🄵被包裝的字典可以作🄵其屬性來存取。

class collections.**UserDict**(*[initialdata]*)

模擬字典的類🄵。實例的🄵容被存於一個字典，可透過*UserDict* 的*data* 屬性來做存取。如果有提供 *initialdata*，*data* 屬性會被初始化🄵其值；要注意指到 *initialdata* 的參照不會被保留，使其可被用於其他目的。

除了支援作🄵對映所需的方法與操作，*UserDict* 實例提供了以下屬性：

data

一個真實的字典，用於儲存*UserDict* 類🄵的資料🄵容。

8.4.8 UserList 物件

此類🄵是 list 物件的包裝器。它是個方便的基礎類🄵，可繼承它🄵覆寫現有方法或加入新方法來定義你所需的一個類似於 list 的類🄵。如此一來，我們可以🄵 list 加入新的特性。

因🄵已經可以直接自*list* 建立子類🄵，這個類🄵的需求已部分被滿足，不過這個類🄵使用起來更方便，因🄵被包裝的 list 可以作🄵其屬性來存取。

class collections.**UserList**(*[list]*)

模擬 list 的類🄵。實例的🄵容被存於一個 list，可透過*UserList* 的*data* 屬性來做存取。實例🄵容被初始化🄵 *list* 的🄵🄵，預設🄵一個空的 list `[]`。*list* 可以是任何 iterable，例如一個真實的 Python list 或是一個*UserList* 物件。

除了支援可變序列的方法與操作，*UserList* 實例提供了以下屬性：

data

一個真實的*list* 物件，用於儲存*UserList* 類🄵的資料🄵容。

子類🄵化的條件： *UserList* 的子類🄵應該要提供一個不需要引數或一個引數的建構函式。回傳一個新序列的 list 操作會從那些實作出來的類🄵建立一個實例，🄵了達成上述目的，它假設建構函式可傳入單一參數來呼叫，該參數即是做🄵數據來源的一個序列物件。

如果希望一個自此獲得的子類🄵不遵從上述要求，那所有該類🄵支援的特殊方法則必須被覆寫；請參考原始碼來理解在這情🄵下哪些方法是必須提供的。

8.4.9 UserString 物件

UserString 類🄵是字串物件的包裝器，因🄵已經可以從*str* 直接建立子類🄵，這個類🄵的需求已經部分被滿足，不過這個類🄵使用起來更方便，因🄵被包裝的字串可以作🄵其屬性來存取。

class collections.**UserString**(*seq*)

模擬字串物件的類🄵。實例的🄵容被存於一個字串物件，可透過*UserString* 的*data* 屬性來做存取。實例🄵容被初始化🄵 *seq* 的🄵🄵，*seq* 引數可以是任何可被🄵建函式*str()* 轉🄵成字串的物件。

除了支援字串的方法和操作以外，*UserString* 實例也提供了以下屬性：

data

一個真實的*str* 物件，用來儲存*UserString* 類🄴的資料🄵容。

在 3.5 版的變更: 新增方法 __getnewargs__、__rmod__、casefold、format_map、isprintable 以及 maketrans。

8.5 collections.abc --- 容器的抽象基底類🄵

在 3.3 版被加入: Formerly, this module was part of the *collections* module.

原始碼： Lib/_collections_abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is *hashable* or whether it is a *mapping*.

An *issubclass()* or *isinstance()* test for an interface works in one of three ways.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):                     # Direct inheritance
    def __init__(self): ...            # Extra method not required by the ABC
    def __getitem__(self, index):  ...  # Required abstract method
    def __len__(self):  ...            # Required abstract method
    def count(self, value): ...        # Optionally override a mixin method
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as "virtual subclasses" of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on *issubclass()* or *isinstance()* tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:                               # No inheritance
    def __init__(self): ...            # Extra method not required by the ABC
    def __getitem__(self, index):  ...  # Abstract method
    def __len__(self):  ...            # Abstract method
    def count(self, value): ...        # Mixin method
    def index(self, value): ...        # Mixin method

Sequence.register(D)                   # Register instead of inherit
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class D does not need to define __contains__, __iter__, and __reversed__ because the in-operator, the *iteration* logic, and the *reversed()* function automatically fall back to using __getitem__ and __len__.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to *None*):

```
class E:
    def __iter__(self): ...
    def __next__(self): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a *Sequence* from a *Mapping*.

在 3.9 版被加入: These abstract classes now support `[]`. See 泛型🄵名型🄵 and **PEP 585**.

8.5.1 Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	抽象方法	Mixin Methods
Container[1]		`__contains__`	
Hashable[1]		`__hash__`	
Iterable[1][2]		`__iter__`	
Iterator[1]	*Iterable*	`__next__`	`__iter__`
Reversible[1]	*Iterable*	`__reversed__`	
Generator[1]	*Iterator*	`send`、`throw`	`close`、`__iter__`、`__next__`
Sized[1]		`__len__`	
Callable[1]		`__call__`	
Collection[1]	*Sized*、*Iterable*、*Container*	`__contains__`、`__iter__`、`__len__`	
Sequence	*Reversible*、*Collection*	`__getitem__`、`__len__`	`__contains__`、`__iter__`、`__reversed__`、`index` 和 `count`
MutableSequence	*Sequence*	`__getitem__`、`__setitem__`、`__delitem__`、`__len__`、`insert`	Inherited *Sequence* methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	*Sequence*	`__getitem__`、`__len__`	*Sequence* 的繼承方法
Set	*Collection*	`__contains__`、`__iter__`、`__len__`	`__le__`、`__lt__`、`__eq__`、`__ne__`、`__gt__`、`__ge__`、`__and__`、`__or__`、`__sub__`、`__rsub__`、`__xor__`、`__rxor__` 和 `isdisjoint`
MutableSet	*Set*	`__contains__`、`__iter__`、`__len__`、`add`、`discard`	Inherited *Set* methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	*Collection*	`__getitem__`、`__iter__`、`__len__`	`__contains__`、`keys`、`items`、`values`、`get`、`__eq__` 和 `__ne__`
MutableMapping	*Mapping*	`__getitem__`、`__setitem__`、`__delitem__`、`__iter__`、`__len__`	Inherited *Mapping* methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	*Sized*		`__init__`、`__len__` 和 `__repr__`
ItemsView	*MappingView*、*Set*		`__contains__`、`__iter__`
KeysView	*MappingView*、*Set*		`__contains__`、`__iter__`
ValuesView	*MappingView*、*Collection*		`__contains__`、`__iter__`
Awaitable[1]		`__await__`	
Coroutine[1]	*Awaitable*	`send`、`throw`	`close`
AsyncIterable[1]		`__aiter__`	
AsyncIterator[1]	*AsyncIterable*	`__anext__`	`__aiter__`
AsyncGenerator[1]	*AsyncIterable*	`asend`、`athrow`	`aclose`、`__aiter__`、`__anext__`
Buffer[1]		`__buffer__`	

[1] These ABCs override *`__subclasshook__()`* to support testing an interface by verifying the required methods are present and have not been set to *None*. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.

[2] Checking `isinstance(obj, Iterable)` detects classes that are registered as *Iterable* or that have an *`__iter__()`* method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

8.5.2 Collections Abstract Base Classes -- Detailed Descriptions

class collections.abc.**Container**

ABC for classes that provide the `__contains__()` method.

class collections.abc.**Hashable**

ABC for classes that provide the `__hash__()` method.

class collections.abc.**Sized**

ABC for classes that provide the `__len__()` method.

class collections.abc.**Callable**

ABC for classes that provide the `__call__()` method.

See 🄵釋 *callable 物件* for details on how to use `Callable` in type annotations.

class collections.abc.**Iterable**

ABC for classes that provide the *`__iter__()`* method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as *`Iterable`* or that have an *`__iter__()`* method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

class collections.abc.**Collection**

ABC for sized iterable container classes.

在 3.6 版被加入.

class collections.abc.**Iterator**

ABC for classes that provide the *`__iter__()`* and *`__next__()`* methods. See also the definition of *iterator*.

class collections.abc.**Reversible**

ABC for iterable classes that also provide the `__reversed__()` method.

在 3.6 版被加入.

class collections.abc.**Generator**

ABC for *generator* classes that implement the protocol defined in **PEP 342** that extends *iterators* with the `send()`, `throw()` and `close()` methods.

See *Annotating generators and coroutines* for details on using `Generator` in type annotations.

在 3.5 版被加入.

class collections.abc.**Sequence**

class collections.abc.**MutableSequence**

class collections.abc.**ByteString**

ABCs for read-only and mutable *sequences*.

Implementation note: Some of the mixin methods, such as *`__iter__()`*, `__reversed__()`, and *`index()`* make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

index(*value*, *start=0*, *stop=None*)

Return first index of *value*.

Raises *`ValueError`* if the value is not present.

Supporting the *start* and *stop* arguments is optional, but recommended.

在 3.5 版的變更: The *`index()`* method gained support for the *stop* and *start* arguments.

Deprecated since version 3.12, will be removed in version 3.17: The *ByteString* ABC has been deprecated.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use *Buffer* or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both *bytes* and *bytearray*. However, since the ABC never had any methods, knowing that an object was an instance of `ByteString` never actually told you anything useful about the object. Other common buffer types such as *memoryview* were also never understood as subtypes of `ByteString` (either at runtime or by static type checkers).

See **PEP 688** for more details.

class collections.abc.**Set**

class collections.abc.**MutableSet**

ABCs for read-only and mutable *sets*.

class collections.abc.**Mapping**

class collections.abc.**MutableMapping**

ABCs for read-only and mutable *mappings*.

class collections.abc.**MappingView**

class collections.abc.**ItemsView**

class collections.abc.**KeysView**

class collections.abc.**ValuesView**

ABCs for mapping, items, keys, and values *views*.

class collections.abc.**Awaitable**

ABC for *awaitable* objects, which can be used in `await` expressions. Custom implementations must provide the `__await__()` method.

Coroutine objects and instances of the *Coroutine* ABC are all instances of this ABC.

> **備🄵**
>
> In CPython, generator-based coroutines (*generators* decorated with *@types.coroutine*) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

在 3.5 版被加入.

class collections.abc.**Coroutine**

ABC for *coroutine* compatible classes. These implement the following methods, defined in coroutine-objects: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All *Coroutine* instances are also instances of *Awaitable*.

> **備🄵**
>
> In CPython, generator-based coroutines (*generators* decorated with *@types.coroutine*) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

See *Annotating generators and coroutines* for details on using `Coroutine` in type annotations. The variance and order of type parameters correspond to those of *Generator*.

在 3.5 版被加入.

class collections.abc.**AsyncIterable**

ABC for classes that provide an `__aiter__` method. See also the definition of *asynchronous iterable*.

在 3.5 版被加入.

class collections.abc.**AsyncIterator**

ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of *asynchronous iterator*.

在 3.5 版被加入.

class collections.abc.**AsyncGenerator**

ABC for *asynchronous generator* classes that implement the protocol defined in **PEP 525** and **PEP 492**.

See *Annotating generators and coroutines* for details on using `AsyncGenerator` in type annotations.

在 3.6 版被加入.

class collections.abc.**Buffer**

ABC for classes that provide the `__buffer__()` method, implementing the buffer protocol. See **PEP 688**.

在 3.12 版被加入.

8.5.3 Examples and Recipes

ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full *Set* API, it is only necessary to supply the three underlying abstract methods: `__contains__()`, *__iter__()*, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and *isdisjoint()*:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using *Set* and *MutableSet* as a mixin:

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an *iterable*. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal *classmethod* called `_from_iterable()` which calls

`cls(iterable)` to produce a new set. If the *Set* mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod or regular method that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

(3) The *Set* mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are *hashable* or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

也參考

- OrderedSet recipe for an example built on *MutableSet*.
- 關於 ABC 的更多資訊請見*abc* module 和 **PEP 3119**。

8.6 `heapq` --- 堆積🄵列 (heap queue) 演算法

原始碼： Lib/heapq.py

這個模組實作了堆積🄴列 (heap queue) 演算法，亦被稱🄵優先🄵列 (priority queue) 演算法。

最小堆積 (min-heap) 是一顆二元樹，樹上每個父節點的值都小於或等於其子節點的值。我們將這種情🄵稱🄴堆積性質不變 (heap invariant)。

對於最小堆積，此實作使用串列，對於所有存在的被比較元素，*k* 都滿足 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]`。元素是從零開始計數。最小堆積有一個有趣的性質：最小的元素永遠在根節點 `heap[0]`。

Max-heaps satisfy the reverse invariant: every parent node has a value *greater* than any of its children. These are implemented as lists for which `maxheap[2*k+1] <= maxheap[k]` and `maxheap[2*k+2] <= maxheap[k]` for all *k* for which the compared elements exist. The root, `maxheap[0]`, contains the *largest* element; `heap.sort(reverse=True)` maintains the max-heap invariant.

`heapq` API 與教科書上的堆積演算法在兩個方面不同：(a) 我們使用從零開始的索引。這使得節點索引和其子節點索引之間的關🄵變得不那🄵明顯，但由於 Python 使用從零開始的索引，所以這樣更適合。(b) 教科書通常專注於最大堆積，因🄵它們適合原地排序。我們的實作偏向於最小堆積，因🄵它們更符合 Python 串列。

這兩個特性使得可以將堆積視🄵一個普通的 Python 串列而不會有意外：`heap[0]` 是最小的元素，而 `heap.sort()` 維持堆積性質不變！

Like *list.sort()*, this implementation uses only the < operator for comparisons, for both min-heaps and max-heaps.

In the API below, and in this documentation, the unqualified term *heap* generally refers to a min-heap. The API for max-heaps is named using a `_max` suffix.

要建立一個堆積，使用初始化🄵 `[]` 的串列，或者分🄵使用*heapify()* 或*heapify_max()* 函式將現有的串列轉🄵🄵最小堆積或最大堆積。

提供了以下針對最小堆積的函式：

`heapq.`**`heappush`**`(`*heap*, *item*`)`

將值 *item* 推入 *heap*，🄵維持最小堆積性質不變。

`heapq.`**`heappop`**`(`*heap*`)`

從 *heap* 取出🄴回傳最小的元素，維持最小堆積性質不變。如果堆積🄵空，會引發*IndexError*。若要在不取出的情🄵下存取最小元素，請使用 `heap[0]`。

heapq.**heappushpop**(*heap*, *item*)

將 *item* 放入 heap，接著從 *heap* 取出🄴回傳最小的元素。這個組合函式比呼叫*heappush()* 之後呼叫*heappop()* 更有效率。

heapq.**heapify**(*x*)

在🄵性時間🄵將串列 *x* 原地轉🄵🄵最小堆積。

heapq.**heapreplace**(*heap*, *item*)

從 *heap* 取出🄴回傳最小的元素，接著將新的 *item* 放進 heap。heap 的大小不會改變。如果 heap 是空的會🄴生*IndexError* 錯誤。

這個一次完成的操作會比呼叫*heappop()* 之後呼叫*heappush()* 更有效率，🄵在維護 heap 的大小不變時更🄵適當，取出/放入的組合函式一定會從 heap 回傳一個元素🄴用 *item* 取代他。

函式的回傳值可能會大於被加入的 *item* 。如果這不是你期望發生的，可以考慮使用*heappushpop()* 替代，他會回傳 heap 的最小值和 *item* 兩個當中比較小的那個，🄵將大的留在 heap 🄵。

提供以下針對最大堆積的函式：

heapq.**heapify_max**(*x*)

在🄵性時間🄵將串列 *x* 原地轉🄵🄵最大堆積。

在 3.14 版被加入.

heapq.**heappush_max**(*heap*, *item*)

將值 *item* 推入最大堆積 *heap*，維持最大堆積性質不變。

在 3.14 版被加入.

heapq.**heappop_max**(*heap*)

從最大堆積 *heap* 取出🄵回傳最大的元素，維持最大堆積性質不變。如果最大堆積🄴空，會引發*IndexError*。若要在不取出的情🄴下存取最大元素，請使用 `maxheap[0]`。

在 3.14 版被加入.

heapq.**heappushpop_max**(*heap*, *item*)

將 *item* 推入 max-heap *heap*，然後取出🄵回傳 *heap* 中最大的元素。這個組合動作比先呼叫*heappush_max()* 再單獨呼叫*heappop_max()* 更有效率。

在 3.14 版被加入.

heapq.**heapreplace_max**(*heap*, *item*)

從最大堆積 *heap* 取出🄴回傳最大的元素，同時推入新的 *item*。最大堆積的大小不會改變。如果最大堆積🄴空，會引發*IndexError*。

The value returned may be smaller than the *item* added. Refer to the analogous function *heapreplace()* for detailed usage notes.

在 3.14 版被加入.

這個模組也提供三個利用 heap 實作的一般用途函式

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)

合🄵多個已排序的輸入🄵🄵生單一且已排序的輸出（舉例：合🄵來自多個 log 檔中有時間戳記的項目）。回傳一個*iterator* 包含已經排序的值。

和 `sorted(itertools.chain(*iterables))` 類似但回傳值是一個 iterable，不會一次把所有資料都放進記憶體中，🄵且假設每一個輸入都已經（由小到大）排序過了。

有兩個選用參數，指定時必須被當作關鍵字參數指定。

key 參數指定了一個*key function* 引數，用來從每一個輸入的元素中🄴定一個比較的依據。預設的值是 `None`（直接比較元素）。

reverse 是一個布林值，如果設定🄴 `True`，則輸入的元素將以相反的比較順序進行合🄵。🄵了達成類似 `sorted(itertools.chain(*iterables), reverse=True)` 的行🄵，所有 iterables 必須由大到小排序。

在 3.5 版的變更: 加入選用參數 *key* 和 *reverse* 。

heapq.**nlargest**(*n*, *iterable*, *key=None*)

回傳一個包含資料 *iterable* 中前 *n* 大元素的 list 。如果有指定 *key* 引數，*key* 會是只有一個引數的函式，用來從每一個在 *iterable* 中的元素提取一個比較的依據（例如 `key=str.lower` ）。效果相當於 `sorted(iterable, key=key, reverse=True)[:n]` 。

heapq.**nsmallest**(*n*, *iterable*, *key=None*)

回傳一個包含資料 *iterable* 中前 *n* 小元素的 list 。如果有指定 *key* 引數，*key* 會是只有一個引數的函式，用來從每一個在 *iterable* 中的元素提取一個比較的依據（例如 `key=str.lower` ）。效果相當於 `sorted(iterable, key=key)[:n]` 。

後兩個函式在 *n* 值比較小時有最好的表現。對於較大的 *n* 值，只用*sorted()* 函式會更有效率。同樣地，當 `n==1` 時，使用🄵建函式*min()* 和*max()* 會有更好的效率。如果需要重🄵使用這些函式，可以考慮將 iterable 轉成真正的 heap 。

8.6.1 基礎范例

堆積排序 (heapsort) 可以透過將所有的值推入一個 heap，🄵且從 heap 中一個接一個彈出最小元素來實作：

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

雖然類似 `sorted(iterable)` ，但跟*sorted()* 不同的是，這個實作不是 stable 的排序。

Heap 中的元素可以是 tuple 。這有利於將要比較的值（例如一個 task 的優先度）和主要資料放在一起排序：

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 優先🄵列實作細節

優先🄵列 (priority queue) 是 heap 的常見用途之一，實作優先🄵列伴隨著下列挑戰：

- 排序的穩定性：如何將兩個擁有相同優先次序 (priority) 的 task 按照他們被加入的順序回傳?
- Tuple 的排序在某些情🄵下會壞掉，例如當 Tuple (priority, task) 的 priorities 相等且 tasks 🄵有一個預設的排序時。
- 當一個 heap 中 task 的 priority 改變時，如何將它移到 heap 正確的位置上?
- 或者一個還🄵被解🄵的 task 需要被🄵除時，要如何從🄵列中找到🄵🄵除指定的 task?

一個針對前兩個問題的解法是：儲存一個包含 priority、entry count 和 task 三個元素的 tuple 。兩個 task 有相同 priority 時，entry count 會讓兩個 task 能根據加入的順序排序。因🄵🄵有任何兩個 task 擁有相同的 entry count，所以永遠不會直接使用 task 做比較。

task 無法比較的另一個解🄵方案是建立一個包裝器類🄵，該類🄵忽略 task 項目，只比較優先等級：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

剩下的問題可以藉由找到要🄵除的 task 🄵更改它的 priority 或者直接將它移除。尋找一個 task 可以使用一個 dictionary 指向🄵列當中的 entry 。

移除 entry 或更改它的 priority 更🄵困難，因🄵這會破壞 heap 的性質。所以一個可行的方案是將原本的 entry 做一個標記表示它已經被🄵除，🄵新增一個擁有新的 priority 的 entry：

```
pq = []                         # 在 heap 中的 entry 串列
entry_finder = {}               # task 對應到 entry 的對映
REMOVED = '<removed-task>'      # 被🄵除的 task 的🄵位器
counter = itertools.count()     # 唯一的序列計數

def add_task(task, priority=0):
    '新增一個 task 或更新一個已存在 task 的 priority'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    '將一個已存在的 task 標記🄵 REMOVED。如果找不到會引發 KeyError。'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    '移除🄵回傳最低 priority 的 task。如果 heap 是空的會引發 KeyError。'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('從空的優先🄵列中 pop 出元素')
```

8.6.3 原理

Heap 是一個陣列對於所有從 0 開始的 index k 都存在性質 `a[k] <= a[2*k+1]` 和 `a[k] <= a[2*k+2]` 。🄵了方便比較，不存在的元素被視🄵無限大。Heap 的一個有趣的性質是：`a[0]` 永遠是最小的元素。

上述乍看之下有些奇怪的不變式，是🄵了實作一個對記憶體來🄵有效率的方法，其表示方式如同錦標賽一般。下列的數字🄵 k，而不是 `a[k]`：

```
                               0

              1                                2

      3               4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

在上面的樹當中，每個單元 k 都會位在 `2*k+1` 與 `2*k+2` 上方。如同體育賽事常見的錦標賽般，每個單元可視🄵其下方兩個單元當中的贏家，我們可以透過追溯整棵樹來找到該贏家曾經對戰過的所有對手。然

而，在許多電腦應用中，我們不需要追溯贏家的完整對戰歷史。F了能更有效率地使用記憶體，當一個贏家F級勝出時，我們用下方較低層級的另一個項目來取代它，至此規則變F一個單元以及它下方兩個單元，包含三個不同項目，但是最上方的單元「勝過」下方兩個單元。

如果能確保滿足這個 heap 的不變式，那F索引 0 顯然是最終的贏家。移除F找到「下一個」贏家最簡單的演算法F：將一個輸家（例如上圖中的單元 30）移動到位置 0，然後從新的位置 0 不斷與下方的位置交F值來向下傳遞，直到滿足不變式F止。這個過程的F雜度顯然是樹的節點數目的對數級F。透過對所有項目F代，可以得到一個F雜度F $O(n \log n)$ 的排序。

這種排序有個好處，只要插入的項目F有「贏過」你最後提取、索引F 0 的元素，你就可以在排序進行的同時有效率地插入新項目。這在模擬情境當中特F有用，其中樹能F保存所有輸入事件，而「贏」意味著最小排程時間。當一個事件排程其它事件的執行時，因這些事件仍在等待進行，所以很容易將它們插入 heap 當中。因此，heap 是一個實現排程器的優秀資料結構（這就是我用以實作 MIDI 編曲器的方法:-）。

多種用於實作排程器的結構現今已被廣泛研究，heap 對此非常有用，因F它們速度相當快，且速度幾乎不受其他因素影響，最壞情F與平均狀F差F無幾。也有其它整體F來更有效率的方法，然而它們的最壞情F可能會非常糟糕。

Heap 在F儲存於硬碟上的大量資料進行排序也非常有用。你可能已經知道，大量資料排序涉及"runs" 的F生（也就是預先排序的序列，其大小通常與 CPU 記憶體的大小有關），之後再對這些 run 合F，而這些合F的過程通常相當巧妙[1]。很重要的一點是，初始排序F生的 run 越長越好。錦標賽是達成這一點的好方法，若你用所有可用記憶體來舉行一場錦標賽，F透過替F與向下交F來處理所有適配目前 run 的值，那F對於隨機F生的輸入，將可以F生長度兩倍於記憶體大小的 run。對於已模糊排序過的輸入，效果更好。

此外，若你將索引F 0 的項目輸出至磁碟，F取得一個無法適配目前錦標賽的輸入（因F該值「勝過」最後的輸出值），則該輸入值就無法插入至 heap 當中，因此 heap 的大小會F小。釋放出來的記憶體可以巧妙地立即再被運用，逐步建構出第二個 heap，其大小增加的速度會與第一個 heap F少的速度一致。當第一個 heap 完全消失時，你可以切F至第二個 heap 開F一個新 run 。這真是個聰明且相當有效率的做法！

總結來F，heap 是值得了解的有用記憶體結構。我在一些應用中使用它們，我認F能有一個'heap' 模組是很棒的。:-)

F解

8.7 `bisect` --- 陣列二分演算法 (Array bisection algorithm)

原始碼： Lib/bisect.py

這個模組維護一個已經排序過的 list ，當我們每次做完插入後不需要再次排序整個 list 。一個很長的 list 的比較操作很花費時間，可以透過F性搜尋或頻繁地詢問來改善。

這個模組被稱F*bisect* 是因F它使用基本二分演算法來完成其工作。不像其它搜尋特定值的二分法工具，本模組中的函式旨在定位插入點。因此，這些函式永遠不會呼叫 `__eq__()` 方法來確認是否找到一個值。相反地，這些函式只呼叫 `__lt__()` 方法，F在陣列中的值回傳一個插入點。

備F

The functions in this module are not thread-safe. If multiple threads concurrently use *bisect* functions on the same sequence, this may result in undefined behaviour. Likewise, if the provided sequence is mutated by a different thread while a *bisect* function is operating on it, the result is undefined. For example, using *insort_left()* on the same list from multiple threads may result in the list becoming unsorted.

此模組提供下面的函式：

[1] 現今的磁碟平衡演算法因F硬碟查找能力而更加F雜難解。在F有查找功能的裝置如大型磁帶機，狀F又不一樣了，人們必須機智地確保（遠遠提前）每次於磁帶上移動都盡可能是最有效率的（也就是盡可能更好地「推進」合F的過程）。有些磁帶甚至能F向後讀取，這也被用來避免倒轉的時間。相信我，真正優秀的磁帶排序看起來相當壯觀！排序一直以來都是一門偉大的藝術！:-)

bisect.**bisect_left**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

在 *a* 當中找到一個位置，讓 *x* 插入後 *a* 仍然是排序好的。參數 *lo* 和 *hi* 用來指定 list 中應該被考慮的子區間，預設是考慮整個 list 。如果 *a* F面已經有 *x* 出現，插入的位置會在所有 *x* 的前面（左邊）。回傳值可以被當作 `list.insert()` 的第一個參數，但列表 *a* 必須先排序過。

回傳的插入點 *ip* 將陣列 *a* 劃分F左右兩個切片，使得對於左切片而言 `all(elem < x for elem in a[lo : ip])` F真，對於右切片而言 `all(elem >= x for elem in a[ip : hi])` F真。

key 可指定一個單一參數的*key function*。函式將套用此 function 在陣列所有元素以得到比較值來計算順位。注意此 function 只會套用在陣列中的元素，不會套用在 *x*。

若 *key* F `None`，元素將直接進行比較，不會呼叫任何鍵函式。

在 3.10 版的變更: 新增 *key* 參數。

bisect.**bisect_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

類似*bisect_left()*，但回傳的插入位置會在所有 *a* 當中的 *x* 的後面（右邊）。

回傳的插入點 *ip* 將陣列 *a* 劃分F左右兩個切片，使得對於左切片而言 `all(elem <= x for elem in a[lo : ip])` F真，對於右切片而言 `all(elem > x for elem in a[ip : hi])` F真。

在 3.10 版的變更: 新增 *key* 參數。

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

將元素 *x* 插入 list *a*，F維持順序。

此函式先使用*bisect_left()* 搜尋插入位置，接著用*insert()* 於 *a* 以將 *x* 插入，F維持添加元素後的順序。

此函式只有在搜尋時會使用 *key* 函式，插入時不會。

注意雖然搜尋是 *O*(log *n*)，但插入是 *O*(*n*)，因此此函式整體時間F雜度是 *O*(*n*)。

在 3.10 版的變更: 新增 *key* 參數。

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

類似*insort_left()*，但插入的位置會在所有 *a* 當中的 *x* 的後面（右邊）。

此函式先使用*bisect_right()* 搜尋插入位置，接著用*insert()* 於 *a* 以將 *x* 插入，F維持添加元素後的順序。

此函式只有在搜尋時會使用 *key* 函式，插入時不會。

注意雖然搜尋是 *O*(log *n*)，但插入是 *O*(*n*)，因此此函式整體時間F雜度是 *O*(*n*)。

在 3.10 版的變更: 新增 *key* 參數。

8.7.1 效能考量

若在需要關注寫入時間的程式當中使用 *bisect()* 和 *insort()*，請特F注意幾個事項:

- 二分法在一段範圍的數值中做搜尋的效率較佳，但若是要存取特定數值，使用字典的表現還是比較好。
- *insort()* 函式的F雜度F *O*(*n*)，因F對數搜尋是以F性時間的插入步驟所主導 (dominate)。
- 搜尋函式F無狀態的 (stateless)，且鍵函式會在使用過後被FF。因此，如果搜尋函式被使用於F圈當中，鍵函式會不斷被重F呼叫於相同的 list 元素。如果鍵函式執行速度不快，請考慮將其以*functools.cache()* 包裝起來以F少重F的計算。另外，也可以透過搜尋預先計算好的鍵列表 (array of precomputed keys) 來定位插入點（如下方範例所示）。

也參考

- 有序容器 (Sorted Collections) 是一個使用 *bisect* 來管理資料之有序集合的高效能模組。

- SortedCollection recipe 使用二分法來建立一個功能完整的集合類🄵 (collection class) 🄵帶有符合直覺的搜尋方法 (search methods) 與支援鍵函式。鍵會預先被計算好，以🄵少搜尋過程中多余的鍵函式呼叫。

8.7.2 搜尋一個已排序的 list

上面的*bisect functions* 在找到數值插入點上很有用，但一般的數值搜尋任務上就不是那🄵的方便。以下的五個函式展示了如何將其轉🄵成標准的有序列表查找函式：

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 范例

`bisect()` 函式可用於數值表中的查找 (numeric table lookup)，這個範例使用`bisect()` 以基於一組有序的數值分界點來🄵一個考試成績找到相對應的字母等級：90 以上是'A'、80 到 89 🄵'B'，依此類推：

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

`bisect()` 與`insort()` 函式也適用於🄵容🄵 tuples（元組）的 lists，*key* 引數可被用以取出在數值表中作🄵排序依據的欄位：

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # Find the first movie released after 1960
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

如果鍵函式會消耗較多運算資源，那可以在預先計算好的鍵列表中搜尋該紀🄵的索引值，以🄵少重🄵的函式呼叫：

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # Or use operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 `array` --- 高效率的數值型陣列

這個模組定義了一個物件型🄵，可以簡潔的表達一個包含基本數值的陣列：字元、整數、浮點數。陣列是一個非常類似 list（串列）的序列型🄵，除了陣列會限制儲存的物件型🄵。在建立陣列時可以使用一個字元的 *type code* 來指定儲存的資料型🄵。以下🄵有被定義的 type codes：

Type code	C Type	Python Type	所需的最小位元組 (bytes)	🄵解
`'b'`	signed char	int	1	
`'B'`	unsigned char	int	1	
`'u'`	wchar_t	Unicode character	2	(1)
`'w'`	Py_UCS4	Unicode character	4	(2)
`'h'`	signed short	int	2	
`'H'`	unsigned short	int	2	
`'i'`	signed int	int	2	
`'I'`	unsigned int	int	2	
`'l'`	signed long	int	4	
`'L'`	unsigned long	int	4	
`'q'`	signed long long	int	8	
`'Q'`	unsigned long long	int	8	
`'f'`	float	float	4	
`'d'`	double	float	8	

🄵解:

(1) 根據平台的不同，它有可能是 16 位元或者 32 位元。

在 3.9 版的變更: 目前 `array('u')` 使用 `wchar_t` 取代已🄵用的 `Py_UNICODE` 作🄵 C type。這個🄵動🄵🄵有影響到它的作用，因🄵自從 Python 3.3 開始 `Py_UNICODE` 即🄵 `wchar_t` 的🄵名。

Deprecated since version 3.3, will be removed in version 3.16: 請改用 `'w'` typecode。

(2) 在 3.13 版被加入.

實際上數值的表示方法是被機器的架構所🄵定（更精准地🄵，被 C 的實作方法🄵定）。實際的大小可以透過*`array.itemsize`* 屬性存取。

這個模組定義了以下項目:

`array.`**`typecodes`**

一個包含所有可用的 type codes 的字串。

這個模組定義了下方的型🄵:

`class` `array.`**`array`**(*typecode*[, *initializer*])

一個新的陣列中的元素被 *typecode* 限制，🄵由選用的 *initializer* 參數初始化，*initializer* 必須是一個*`bytes`* 或*`bytearray`* 物件、一個 Unicode 字串或包含適當型🄵元素的可🄵代物件 (iterable)。

如果給定的是一個*`bytes`* 或*`bytearray`* 物件，新的陣列初始化時會傳入*`frombytes()`* 方法；如🄵 Unicode 字串則會傳入*`fromunicode()`* 方法；其他情🄵時，一個 initializer 的可🄵代物件將被傳入*`extend()`* 方法之中來將初始項目新增至陣列。

陣列支援常見的序列操作，包含索引 (indexing)、切片 (slicing)、串接 (concatenation)、相乘 (multiplication) 等。當使用切片進行賦值時，賦值的陣列必須具備相同的 type code，其他型🄵的數值將導致*`TypeError`*。陣列同時也實作了緩衝區介面，可以在任何支援*bytes-like objects* 的地方使用。

引發稽核事件 *(auditing event)* `array.__new__` 🄵附帶引數 `typecode`、`initializer`。

`typecode`

typecode 字元被用在建立陣列時。

`itemsize`

陣列當中的一個元素在🄵部需要的位元組長度。

`append`(*x*)

新增一個元素 *x* 到陣列的最尾端。

buffer_info()

回傳一個 tuple `(address, length)` 表示目前的記憶體位置和陣列儲存元素的緩衝區記憶體長度。緩衝區的長度單位是位元組，🄵可以用 `array.buffer_info()[1] * array.itemsize` 計算得到。這偶爾會在底層操作需要記憶體位置的輸出輸入時很有用，例如 `ioctl()` 指令。只要陣列存在且🄵有使用任何更改長度的操作時，回傳的數值就有效。

> **備🄵**
>
> 當使用來自 C 或 C++ 程式碼（這是唯一使得這個資訊有效的途徑）的陣列物件時，更適當的做法是使用陣列物件支援的緩衝區介面。這個方法維護了向後兼容性，🄵應該在新的程式碼中避免。關於緩衝區介面的文件在 bufferobjects。

byteswap()

”Byteswap” 所有陣列中的物件。這只有支援物件長度🄵 1、2、4 或 8 位元組的陣列，其他型🄵的值會導致*RuntimeError*。這在從機器讀取位元順序不同的檔案時很有用。

count(*x*)

回傳 *x* 在陣列中出現了幾次。

extend(*iterable*)

從 *iterable* 中新增元素到陣列的尾端，如果 *iterable* 是另一個陣列，它必須有完全相同的 type code，如果不同會導致*TypeError*。如果 *iterable* 不是一個陣列，它必須可以被🄵代 (iterable) 且其中的元素必須是可以被加入陣列中的正確型🄵。

frombytes(*buffer*)

從*bytes-like object* 中新增元素。讀取時會將其🄵容當作一個機器數值組成的陣列（就像從檔案中使用*fromfile()* 方法讀出的資料）。

在 3.2 版被加入: 將 `fromstring()` 更名🄵*frombytes()*，使其更加清晰易懂。

fromfile(*f*, *n*)

從*file object* *f* 讀取 *n* 個元素（作🄵機器數值），接著將這些元素加入陣列的最尾端。如果只有少於 *n* 個有效的元素會導致*EOFError*，但有效的元素仍然會被加入陣列中。

fromlist(*list*)

從 list 中新增元素。這等價於 `for x in list: a.append(x)`，除了有型🄵錯誤🄵生時，陣列會保持原狀不會被更改。

fromunicode(*s*)

用給定的 Unicode 字串擴展這個陣列。陣列的 type code 必須是 `u` 或 `'w'`；其他的型🄵會導致*ValueError* 被引發。使用 `array.frombytes(unicodestring.encode(enc))` 來新增 Unicode 資料到一個其他型🄵的陣列。

index(*x*[, *start*[, *stop*]])

回傳 *i* 的最小數值，使得 *i* 成🄵陣列之中第一次出現 *x* 的索引。選擇性的引數 *start* 及 *stop* 則可以被用來在指定的陣列空間中搜尋 *x*。如果 *x* 不存在將導致*ValueError*。

在 3.10 版的變更: 新增選擇性的參數 *start* 及 *stop*。

insert(*i*, *x*)

在位置 *i* 之前插入一個元素 *x*。負數的索引值會從陣列尾端開始數。

pop([*i*])

移除🄵回傳陣列索引值 *i* 的元素。選擇性的引數 *i* 預設🄵 `-1`，所以預設會🄵除🄵回傳最後一個元素。

remove(*x*)

從陣列中🄵除第一個出現的 *x*。

clear()
從陣列中🄴除所有元素。
在 3.13 版被加入.

reverse()
反轉陣列中元素的順序。

tobytes()
將陣列轉🄴另一個機器數值組成的陣列🄴回傳它的位元組表示（跟用*tofile()* 方法寫入檔案時的位元序列相同）。
在 3.2 版被加入: 🄴了明確性，過去的 `tostring()` 已更名🄴*tobytes()*。

tofile(*f*)
將所有元素（作🄴機器數值）寫入*file object* *f*。

tolist()
不更改元素，將陣列轉🄴一般的 list。

tounicode()
將陣列轉🄴一個 Unicode 字串。陣列的型🄴必須🄴 `u` 或 `'w'`；其他型🄴的陣列會引發*ValueError*。請使用 `array.tobytes().decode(enc)` 來🄵其他型🄴的陣列轉🄴 Unicode 字串。

陣列物件的字串表示形式🄴 `array(typecode, initializer)`。若🄵空陣列則參數 *initializer* 被省略，若 *typecode* 是 `'u'` 或 `'w'` 將被表示🄴 Unicode 字串，其他情🄵則被表示🄵由數字組成的 list。只要*array* class（類🄵）透過 `from array import array` 的方式引入，便能確保該字串表示能透過*eval()* 轉🄵回一個擁有相同型🄵及數值的陣列。如果它包含相應的浮點值，`inf` 和 `nan` 也必須被定義。範例：

```
array('l')
array('w', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

也參考

***struct* 模組**
將包含不同資料類型的二進位資料包裝與解開包裝。

NumPy
NumPy 套件定義了另一個陣列型🄵

8.9 `weakref` --- 弱參照

原始碼： Lib/weakref.py

---

weakref 模組允許 Python 程式設計師建立對物件的 *弱參照*。

在以下文章中，術語 *參照目標 (referent)* 表示被弱參照所參考的物件。

對物件的弱參照不足以使物件保持存在：當對參照目標的唯一剩下的參照是弱參照時，*garbage collection* 可以自由地銷🄴參照目標🄴將其記憶體重新用於其他用途。然而，在物件被確實銷🄴之前，即使🄵有對該物件的🄵參照 (strong reference)，弱參照也可能會回傳該物件。

弱參照的主要用途是實作保存大型物件的快取或對映，其不希望大型物件僅僅因🄴它出現在快取或對映中而保持存在。

例如，如果你有許多大型的二進位影像物件，你可能會想要🄴每個物件關聯 (associate) 一個名稱。如果你使用 Python 字典將名稱對映到影像，或將影像對映到名稱，則影像物件將保持存活，僅因🄴它們在字

典中作🄵值 (value) 或鍵 (key) 出現。*weakref* 模組提供的*WeakKeyDictionary* 和*WeakValueDictionary* 類🄵是另一種選擇，它們使用弱參照來建構對映，這些對映不會僅因🄵物件出現在對映物件中而使物件保持存活。例如，如果一個影像物件是*WeakValueDictionary* 中的一個值，那🄵當對該影像物件最後的參照是弱對映 (weak mapping) 所持有的弱參照時，垃圾回收 (garbage collection) 可以回收該物件，且其對應的條目在弱對映中會被完全地🄵除。

WeakKeyDictionary 和*WeakValueDictionary* 在其實作中使用弱參照，在弱參照上設定回呼函式，此弱參照在垃圾回收取回鍵或值時通知弱字典。*WeakSet* 實作了*set* 介面，但保留對其元素的弱參照，就像*WeakKeyDictionary* 一樣。

finalize 提供了一種直接的方法來🄵🄵在物件被垃圾回收時呼叫的清理函式。這比在原始弱參照上設定回呼函式更容易使用，因🄵模組在物件被回收前會自動確保最終化器 (finalizer) 保持存活。

大多數程式應該發現使用這些弱容器種類之一或*finalize* 就足🄵了—通常不需要直接建立自己的弱參照。低層級的機制由*weakref* 模組公開，以利於進階用途。

🄵非所有物件都可以被弱參照。支援弱參照的物件包括類🄵實例、用 Python （但不是 C）編寫的函式、實例方法、集合、凍結集合 (frozenset)、一些檔案物件、🄵生器、類型物件、socket、陣列、雙向🄵列、正規表示式模式物件和程式碼物件。

在 3.2 版的變更: 新增了對 thread.lock、threading.Lock 和程式碼物件的支援。

一些🄵建型🄵，例如*list* 和*dict* 不直接支援弱參照，但可以透過子類🄵化來支援：

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # 這個物件是可被弱參照的
```

其他🄵建型🄵，例如*tuple* 和*int* 即使在子類🄵化時也不支援弱參照。

擴充型🄵 (extension type) 可以輕易地支援弱參照；請參🄵 weakref-support。

當🄵給定的型🄵定義 __slots__ 時，弱參照支援將被停用，除非 '__weakref__' 字串也存在於 __slots__ 宣告的字串序列中。詳情請參🄵 __slots__ 文件。

class weakref.**ref**(*object*[, *callback*])

回傳對 *object* 的弱參照。如果參照目標仍存活，則可以透過呼叫參照物件來取回原始物件；如果參照目標已不存活，呼叫參照物件將導致*None* 被回傳。如果 *callback* 被提供而非*None*，且回傳的弱參照物件仍存活，那🄵當物件即將被最終化 (finalize) 時，回呼將被呼叫；弱參照物件將作🄵唯一的參數傳遞給回呼；參照物件將不再可用。

🄵同一個物件建構多個弱參照是可行的。🄵每個弱參照🄵🄵的回呼將按照最新到最舊🄵🄵的回呼順序來被呼叫。

回呼引發的例外將在標准錯誤輸出中被🄵明，但無法被傳播；它們的處理方式與物件的 __del__() 方法引發的例外完全相同。

如果 *object* 是可雜🄵的，那🄵弱參照就是可雜🄵的。即使在 *object* 被🄵除後，它們仍將保留其雜🄵值。如果僅在 *object* 🄵除後才第一次呼叫*hash()*，則該呼叫將引發*TypeError*。

弱參照支援相等性的測試，但不支援排序。如果參照目標仍存活，則兩個參照與其參照目標具有相同的相等關🄵（無論 *callback* 如何）。如果任一參照目標已被🄵除，則僅當參照物件是同一物件時，參照才相等。

這是一個可子類🄵化的型🄵，而不是一個工廠函式。

__callback__

此唯讀屬性回傳目前與弱參照關聯的回呼。如果🄵有回呼或弱參照的參照目標已不存活，那🄵該屬性的值🄵 None。

在 3.4 版的變更: 新增*__callback__* 屬性。

weakref.**proxy**(*object*[, *callback*])

回傳一個使用弱參照的 *object* 的代理 (proxy)。這支援在大多數情境中使用代理，而不需要對弱參照物件明確地取消參照。回傳的物件將具有 ProxyType 或 CallableProxyType 型🄵，具體取🄵於

object 是否🄵可呼叫物件。無論參照目標如何，代理物件都不是*hashable*；這避免了與其基本可變物件本質相關的許多問題，🄵阻止它們作🄵字典的鍵被使用。*callback* 與`ref()` 函式的同名參數是相同的。

在參照目標被垃圾回收後存取代理物件的屬性會引發`ReferenceError`。

在 3.8 版的變更: 提供對代理物件的運算子支援，以包括矩陣乘法運算子 `@` 和 `@=`。

`weakref.`**`getweakrefcount`**`(`*object*`)`

回傳參照 *object* 的弱參照和代理的數量。

`weakref.`**`getweakrefs`**`(`*object*`)`

回傳參照 *object* 的所有弱參照和代理物件的一個串列。

`class` `weakref.`**`WeakKeyDictionary`**`(`[*dict*]`)`

弱參照鍵的對映類🄵。當不再有對鍵的🄵參照時，字典中的條目將被🄵🄵。這可用於將附加資料與應用程式其他部分擁有的物件相關聯，而無需向這些物件新增屬性。這對於覆蓋屬性存取的物件特🄵有用。

請注意，當將與現有鍵具有相同值的鍵（但識🄵性不相等）插入字典時，它會替🄵該值，但不會替🄵現有鍵。因此，當🄵除對原始鍵的參照時，它也會🄵除字典中的條目：

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

一個變通的解法是在重新賦值 (reassignment) 之前🄵除鍵：

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2   # d = {k2: 2}
>>> del k1      # d = {k2: 2}
```

在 3.9 版的變更: 新增對 `|` 和 `|=` 運算子的支持，如 **PEP 584** 中所🄵明。

`WeakKeyDictionary` 物件有一個直接公開🄵部參照的附加方法。參照在被使用時不保證是" 存活的"，因此在使用之前需要檢查呼叫參照的結果。這可以用來防止建立會導致垃圾回收器保留鍵的時間超過其所需時間的參照。

`WeakKeyDictionary.`**`keyrefs`**`()`

回傳對鍵的弱參照的可🄵代物件。

`class` `weakref.`**`WeakValueDictionary`**`(`[*dict*]`)`

弱參照值的對映類🄵。當不再存在對值的🄵參照時，字典中的條目將被🄵🄵。

在 3.9 版的變更: 新增對 `|` 和 `|=` 運算子的支持，如 **PEP 584** 中所🄵明。

`WeakValueDictionary` 物件有一個附加方法，它與`WeakKeyDictionary.keyrefs()` 方法有相同的問題。

`WeakValueDictionary.`**`valuerefs`**`()`

回傳對值的弱參照的可🄵代物件。

`class` `weakref.`**`WeakSet`**`(`[*elements*]`)`

保留對其元素的弱參照的集合類🄵。當不再存在對某個元素的🄵參照時，該元素將被🄵🄵。

class weakref.**WeakMethod**(*method*[, *callback*])

一個特🄵的*ref* 子類🄵，其模擬對🄴結方法 (bound method)（即在類🄵上定義🄴在實例上查找的方法）的弱參照。由於🄴結方法是短暫存在的，因此標准弱參照無法保留它。*WeakMethod* 有特殊的程式碼來重新建立🄵結方法，直到物件或原始函式死亡：

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback 與*ref()* 函式的同名參數是相同的。

在 3.4 版被加入.

class weakref.**finalize**(*obj*, *func*, */*, **args*, ***kwargs*)

回傳可呼叫的最終化器物件，此物件在 *obj* 被垃圾回收時會被呼叫。與一般的弱參照不同，最終化器將始終存在，直到參照物件被回收🄵止，從而大大簡化了生命🄵期管理。

最終化器在被呼叫（明確呼叫或在垃圾回收時）之前被視🄵*存活*，之後它就會*死亡*。呼叫存活的最終化器會回傳 `func(*arg, **kwargs)` 的計算結果，而呼叫死亡的最終化器會回傳*None*。

垃圾回收期間最終化器回呼引發的例外會在標准錯誤輸出中顯示，但無法傳播。它們的處理方式與從物件的 `__del__()` 方法或弱參照的回呼引發的例外相同。

當程式結束時，除非該最終化器的*atexit* 屬性已被設定🄵 false，否則每個存活的最終化器會被呼叫。它們以與建立相反的順序被呼叫。

當模組的 globals 可能被*None* 取代時，最終化器永遠不會在*interpreter shutdown* 的後期叫用（invoke）其回呼。

__call__()

如果 *self* 仍存活，則將其標記🄵死亡🄵回傳呼叫 `func(*args, **kwargs)` 的結果。如果 *self* 已死亡，則回傳*None*。

detach()

如果 *self* 仍存活，則將其標記🄵死亡🄵回傳元組 `(obj, func, args, kwargs)`。如果 *self* 已死亡，則回傳*None*。

peek()

如果 *self* 仍存活，則回傳元組 `(obj, func, args, kwargs)`。如果 *self* 已死亡，則回傳*None*。

alive

如果最終化器仍存活，則屬性🄵 true，否則🄵 false。

atexit

一個可寫的布林屬性，預設🄵 true。當程式結束時，它會呼叫*atexit* 🄵 true 的所有剩余且仍存活的最終化器。它們以與建立相反的順序被呼叫。

> **備🄵**
>
> 確保 *func*、*args* 和 *kwargs* 不直接或間接擁有對 *obj* 的任何參照非常重要，否則 *obj* 將永遠不會被垃圾回收。尤其 *func* 不應該是 *obj* 的🄵結方法。

在 3.4 版被加入.

`weakref.`**`ReferenceType`**
弱參照物件的型🄵物件。

`weakref.`**`ProxyType`**
非可呼叫物件的代理的型🄵物件。

`weakref.`**`CallableProxyType`**
可呼叫物件的代理的型🄵物件。

`weakref.`**`ProxyTypes`**
包含代理的所有型🄵物件的序列。這可以讓測試物件是否🄵代理變得更簡單，而無需依賴命名兩種代理型🄵。

> **也參考**
>
> **PEP 205 - 弱參照**
> 此功能的提案和理由，包括早期實作的連結以及其他語言中類似功能的資訊。

8.9.1 弱參照物件

弱參照物件除了*ref.__callback__* 之外🄵有任何方法和屬性。弱參照物件允許透過呼叫來取得參照目標（如果它仍然存在）：

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果參照目標不再存活，則呼叫參照物件將回傳*None*：

```
>>> del o, o2
>>> print(r())
None
```

應該使用運算式 `ref() is not None` 來測試弱參照物件是否仍然存活。需要使用參照物件的應用程式程式碼通常應遵循以下模式：

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

使用對「活性 (liveness)」的單獨測試會在執行緒應用程式中建立競🄴條件 (race condition)；另一個執行緒可能在弱參照被呼叫之前讓該弱參照失效；上方顯示的慣用作法在執行緒應用程式和單執行緒應用程式中都是安全的。

可以透過子類🄴化來建立*ref* 物件的特殊版本。這在*WeakValueDictionary* 的實作中被使用，以🄴少對映中每個條目的記憶體開銷。這對於將附加資訊與參照相關聯最有用，但也可用於在呼叫上插入附加處理以檢索參照目標。

這個範例展示如何使用*ref* 的子類🄴來儲存有關物件的附加資訊🄴影響存取參照目標時回傳的值：

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 范例

這個簡單的範例展示了應用程式如何使用物件 ID 來檢索它以前見過 的物件。物件的 ID 之後可以在其他資料結構中使用，而不必🄴制物件保持存活，但如果這樣做，仍然可以透過 ID 檢索物件。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 最終化器物件

使用*finalize* 的最大優點是可以輕松🄵🄵回呼，而無需保留回傳的最終化器物件。例如

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

最終化器也可以直接被呼叫。然而，最終化器最多會叫用回呼一次。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

你可以使用最終化器的*detach()* 方法來取消🄵🄵最終化器。這會殺死最終化器🄵回傳建立建構函式時傳遞給建構函式的引數。

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

除非你將*atexit* 屬性設🄵*False*，否則當程式結束時，最終化器將會被呼叫如果其仍然存在。例如

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 最終化器與 __del__() 方法的比較

假設我們要建立一個類🄴，其實例代表臨時目🄵。當以下任一事件發生時，應🄴除目🄵及其🄴容：

- 該物件被垃圾回收，
- 該物件的 remove() 方法被呼叫，或者
- 程式結束。

我們可以用以下的方式來嘗試使用 __del__() 方法實作該類🄴：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

從 Python 3.4 開始，`__del__()` 方法不再阻止參照循環 (reference cycle) 被垃圾回收，🄵且在*interpreter shutdown* 期間不再🄵制將模組的 globals 設🄵`None`。所以這段程式碼在 CPython 上應該可以正常運作。

然而，🄵所周知，對 `__del__()` 方法的處理是特地實作的，因🄵它依賴於直譯器的垃圾回收器實作的🄵部細節。

更耐用的替代方案可以是定義一個最終化器，其僅參照需要的特定函式和物件，而不是存取物件的完整狀態：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

定義如下，我們的最終化器僅接收對適當清理目🄵所需的詳細資訊的參照。如果物件從未被垃圾回收，則最終化器仍將在結束時被呼叫。

基於 weakref 的最終化器的另一個優點是它們可用於🄵定義由第三方控制的類🄵🄵🄵最終化器，例如在卸載模組時執行程式碼：

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

備🄵

如果在程式結束時在常駐的 (daemonic) 執行緒中建立最終化器物件，則最終化器有可能在結束時不會被呼叫。然而，在常駐的執行緒中*atexit.register()*、`try: ... finally: ...` 和 `with: ...` 也不保證清理會發生。

8.10 `types` --- 動態型🄵建立與🄵建型🄵名稱

原始碼： Lib/types.py

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like *int* or *str* are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.10.1 Dynamic Type Creation

`types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: None`.

在 3.3 版被加入.

`types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

在 3.3 版被加入.

在 3.6 版的變更: The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method.

也參考

metaclasses
Full details of the class creation process supported by these functions

PEP 3115 - Metaclasses in Python 3000
Introduced the `__prepare__` namespace hook

`types.`**`resolve_bases`**(*bases*)

Resolve MRO entries dynamically as specified by **PEP 560**.

This function looks for items in *bases* that are not instances of *type*, and returns a tuple where each such object that has an `__mro_entries__()` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of *type*, or it doesn't have an `__mro_entries__()` method, then it is included in the return tuple unchanged.

在 3.7 版被加入.

`types.`**`get_original_bases`**(*cls*, */*)

Return the tuple of objects originally given as the bases of *cls* before the `__mro_entries__()` method has been called on any bases (following the mechanisms laid out in **PEP 560**). This is useful for introspecting *Generics*.

For classes that have an `__orig_bases__` attribute, this function returns the value of `cls.__orig_bases__`. For classes without the `__orig_bases__` attribute, `cls.__bases__` is returned.

舉例來🄵:

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)
```

(繼續下一頁)

(繼續上一頁)

```
assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

在 3.12 版被加入.

也參考

PEP 560 - Core support for typing module and generic types

8.10.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for *`isinstance()`* or *`issubclass()`* checks.

If you instantiate any of these types, note that signatures may vary between Python versions.

Standard names are defined for the following types:

`types.`**`NoneType`**

The type of *`None`*.

在 3.10 版被加入.

`types.`**`FunctionType`**

`types.`**`LambdaType`**

The type of user-defined functions and functions created by `lambda` expressions.

引發一個附帶引數 `code` 的稽核事件 `function.__new__`。

The audit event only occurs for direct instantiation of function objects, and is not raised for normal compilation.

`types.`**`GeneratorType`**

The type of *generator*-iterator objects, created by generator functions.

`types.`**`CoroutineType`**

The type of *coroutine* objects, created by `async def` functions.

在 3.5 版被加入.

`types.`**`AsyncGeneratorType`**

The type of *asynchronous generator*-iterator objects, created by asynchronous generator functions.

在 3.6 版被加入.

`class` `types.`**`CodeType`**(***kwargs*)

The type of code objects such as returned by *`compile()`*.

引發一個附帶引數 `code`、`filename`、`name`、`argcount`、`posonlyargcount`、`kwonlyargcount`、`nlocals`、`stacksize`、`flags` 的稽核事件 `code.__new__`。

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

types.**CellType**

The type for cell objects: such objects are used as containers for a function's *closure variables*.

在 3.8 版被加入.

types.**MethodType**

The type of methods of user-defined class instances.

types.**BuiltinFunctionType**

types.**BuiltinMethodType**

The type of built-in functions like *len()* or *sys.exit()*, and methods of built-in classes. (Here, the term "built-in" means "written in C".)

types.**WrapperDescriptorType**

The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

在 3.7 版被加入.

types.**MethodWrapperType**

The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

在 3.7 版被加入.

types.**NotImplementedType**

The type of *NotImplemented*.

在 3.10 版被加入.

types.**MethodDescriptorType**

The type of methods of some built-in data types such as *str.join()*.

在 3.7 版被加入.

types.**ClassMethodDescriptorType**

The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

在 3.7 版被加入.

class types.**ModuleType**(*name*, *doc=None*)

The type of *modules*. The constructor takes the name of the module to be created and optionally its *docstring*.

> **也參考**
>
> **Documentation on module objects**
>
> Provides details on the special attributes that can be found on instances of `ModuleType`.
>
> ***importlib.util.module_from_spec()***
>
> Modules created using the `ModuleType` constructor are created with many of their special attributes unset or set to default values. `module_from_spec()` provides a more robust way of creating `ModuleType` instances which ensures the various attributes are set appropriately.

types.**EllipsisType**

The type of *Ellipsis*.

在 3.10 版被加入.

class types.**GenericAlias**(*t_origin*, *t_args*)

The type of *parameterized generics* such as `list[int]`.

`t_origin` should be a non-parameterized generic class, such as `list`, `tuple` or `dict`. `t_args` should be a *tuple* (possibly of length 1) of types which parameterize `t_origin`:

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

在 3.9 版被加入.

在 3.9.2 版的變更: This type can now be subclassed.

也參考

Generic Alias Types
In-depth documentation on instances of `types.GenericAlias`

PEP 585 - Type Hinting Generics In Standard Collections
Introducing the `types.GenericAlias` class

class `types.`**UnionType**

The type of *union type expressions*.

在 3.10 版被加入.

在 3.14 版的變更: This is now an alias for *typing.Union*.

class `types.`**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

The type of traceback objects such as found in `sys.exception().__traceback__`.

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

`types.`**FrameType**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

`types.`**GetSetDescriptorType**

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the *property* type, but for classes defined in extension modules.

`types.`**MemberDescriptorType**

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the *property* type, but for classes defined in extension modules.

In addition, when a class is defined with a `__slots__` attribute, then for each slot, an instance of `MemberDescriptorType` will be added as an attribute on the class. This allows the slot to appear in the class's `__dict__`.

CPython 實作細節: In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class `types.`**MappingProxyType**(*mapping*)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

在 3.3 版被加入.

在 3.9 版的變更: Updated to support the new union (`|`) operator from **PEP 584**, which simply delegates to the underlying mapping.

key in proxy

Return `True` if the underlying mapping has a key *key*, else `False`.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a *KeyError* if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(*key*[, *default*])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a *KeyError*.

items()

Return a new view of the underlying mapping's items (`(key, value)` pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

reversed(proxy)

Return a reverse iterator over the keys of the underlying mapping.

在 3.9 版被加入.

hash(proxy)

Return a hash of the underlying mapping.

在 3.12 版被加入.

class types.**CapsuleType**

The type of capsule objects.

在 3.13 版被加入.

8.10.3 Additional Utility Classes and Functions

class types.**SimpleNamespace**

A simple *object* subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike *object*, with `SimpleNamespace` you can add and remove attributes.

SimpleNamespace objects may be initialized in the same way as *dict*: either with keyword arguments, with a single positional argument, or with both. When initialized with keyword arguments, those are directly added to the underlying namespace. Alternatively, when initialized with a positional argument, the underlying namespace will be updated with key-value pairs from that argument (either a mapping object or an *iterable* object producing key-value pairs). All such keys must be strings.

The type is roughly equivalent to the following code:

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other, SimpleNamespace):
           return self.__dict__ == other.__dict__
        return NotImplemented
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use *`namedtuple()`* instead.

`SimpleNamespace` objects are supported by *`copy.replace()`*.

在 3.3 版被加入.

在 3.9 版的變更: Attribute order in the repr changed from alphabetical to insertion (like `dict`).

在 3.13 版的變更: Added support for an optional positional argument.

`types.`**`DynamicClassAttribute`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's __getattr__ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see *`enum.Enum`* for an example).

在 3.4 版被加入.

8.10.4 Coroutine Utility Functions

`types.`**`coroutine`**(*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of *`collections.abc.Generator`*, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

在 3.5 版被加入.

8.11 copy --- 淺層 (shallow) 和深層 (deep) 🄵🄵操作

原始碼： Lib/copy.py

Python 的賦值陳述式不🄵🄵物件，而是建立目標和物件的🄵結 (binding) 關🄵。對於可變 (mutable) 或包含可變項目 (mutable item) 的集合，有時會需要一份副本來改變特定副本，而不必改變其他副本。本模組提供了通用的淺層🄵🄵和深層🄵🄵操作（如下所述）。

介面摘要：

copy.**copy**(*obj*)
回傳 *obj* 的淺層🄵🄵。

copy.**deepcopy**(*obj*[, *memo*])
回傳 *obj* 的深層🄵🄵。

copy.**replace**(*obj*, */*, ***changes*)
Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

在 3.13 版被加入.

exception copy.**Error**
引發針對特定模組的錯誤。

淺層與深層🄵🄵的區🄵僅與🄵合物件（即包含 list 或類🄵的實例等其他物件的物件）相關：

- 淺層🄵🄵建構一個新的🄵合物件，然後（在🄵可能的範圍🄵）將原始物件中找到的物件的參照插入其中。
- 深層🄵🄵建構一個新的🄵合物件，然後遞🄵地將在原始物件🄵找到的物件的副本插入其中。

深層🄵🄵操作通常存在兩個問題，而淺層🄵🄵操作🄵不存在這些問題：

- 遞🄵物件（直接或間接包含對自身參照的🄵合物件）可能會導致遞🄵🄵圈。
- 由於深層🄵🄵會🄵🄵所有🄵容，因此可能會有過多🄵🄵（例如應該在副本之間共享的資料）。

deepcopy() 函式用以下方式避免了這些問題：

- 保留在目前🄵🄵過程中已🄵🄵的物件的 memo 字典；以及
- 允許使用者定義的類🄵🄵寫 (override) 🄵🄵操作或🄵🄵的元件集合。

該模組不🄵🄵模組、方法、堆🄵追🄵（stack trace）、堆🄵框（stack frame）、檔案、socket、視窗、陣列以及任何類似的型🄵。它透過不變更原始物件🄵將其回傳來（淺層或深層地）" 🄵🄵" 函式和類🄵；這與*pickle* 模組處理這類問題的方式是相似的。

字典的淺層🄵🄵可以使用*dict.copy()*，而 list 的淺層🄵🄵可以透過賦值整個 list 的切片 (slice) 完成，例如，`copied_list = original_list[:]`。

類🄵可以使用與操作 pickle 相同的介面來控制🄵🄵操作，關於這些方法的描述資訊請參考*pickle* 模組。實際上，*copy* 模組使用的正是從*copyreg* 模組中🄵🄵的 pickle 函式。

In order for a class to define its own copy implementation, it can define special methods *__copy__()* and *__deepcopy__()*.

object.**__copy__**(*self*)
Called to implement the shallow copy operation; no additional arguments are passed.

object.**__deepcopy__**(*self*, *memo*)
用來實作深層🄵🄵操作；它會傳遞一個引數，即 *memo* 字典。如果 __deepcopy__ 實現需要建立一個元件的深層🄵🄵，它應當呼叫*deepcopy()* 函式🄵以該元件作🄵第一個引數、以該 *memo* 字典作🄵第二個引數。*memo* 字典應當被當作不透明物件 (opaque object) 來處理

Function `copy.replace()` is more limited than *copy()* and *deepcopy()*, and only supports named tuples created by *namedtuple()*, *dataclasses*, and other classes which define method *__replace__()*.

object.**__replace__**(*self*, */*, ***changes*)
This method should create a new object of the same type, replacing fields with values from *changes*.

在 3.13 版被加入.

也參考

***pickle* 模組**
支援物件之狀態檢索 (state retrieval) 和恢復 (restoration) 相關特殊方法的討論。

8.12 pprint --- 資料美化列印器

原始碼： Lib/pprint.py

The *pprint* module provides a capability to ”pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width, adjustable by the *width* parameter defaulting to 80 characters.

Dictionaries are sorted by key before the display is computed.

在 3.9 版的變更: Added support for pretty-printing *types.SimpleNamespace*.

在 3.10 版的變更: Added support for pretty-printing *dataclasses.dataclass*.

8.12.1 函式

pprint.**pp**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=False*, *underscore_numbers=False*)

Prints the formatted representation of *object*, followed by a newline. This function may be used in the interactive interpreter instead of the *print()* function for inspecting values. Tip: you can reassign `print = pprint.pp` for use within a scope.

參數

- **object** -- 將被印出的物件。
- **stream** (*file-like object* | None) -- A file-like object to which the output will be written by calling its `write()` method. If `None` (the default), *sys.stdout* is used.
- **indent** (`int`) -- The amount of indentation added for each nesting level.
- **width** (`int`) -- The desired maximum number of characters per line in the output. If a structure cannot be formatted within the width constraint, a best effort will be made.
- **depth** (`int` | *None*) -- The number of nesting levels which may be printed. If the data structure being printed is too deep, the next contained level is replaced by `...`. If `None` (the default), there is no constraint on the depth of the objects being formatted.
- **compact** (`bool`) -- Control the way long *sequences* are formatted. If `False` (the default), each item of a sequence will be formatted on a separate line, otherwise as many items as will fit within the *width* will be formatted on each output line.
- **sort_dicts** (`bool`) -- If `True`, dictionaries will be formatted with their keys sorted, otherwise they will be displayed in insertion order (the default).
- **underscore_numbers** (`bool`) -- If `True`, integers will be formatted with the _ character for a thousands separator, otherwise underscores are not displayed (the default).

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

在 3.8 版被加入.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Alias for *pp()* with *sort_dicts* set to `True` by default, which would automatically sort the dictionaries' keys, you might want to use *pp()* instead where it is `False` by default.

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Return the formatted representation of *object* as a string. *indent*, *width*, *depth*, *compact*, *sort_dicts* and *underscore_numbers* are passed to the *PrettyPrinter* constructor as formatting parameters and their meanings are as described in the documentation above.

pprint.**isreadable**(*object*)

Determine if the formatted representation of *object* is "readable", or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Determine if *object* requires a recursive representation. This function is subject to the same limitations as noted in *saferepr()* below and may raise an *RecursionError* if it fails to detect a recursive object.

pprint.**saferepr**(*object*)

Return a string representation of *object*, protected against recursion in some common data structures, namely instances of *dict*, *list* and *tuple* or subclasses whose `__repr__` has not been overridden. If the representation of object exposes a recursive entry, the recursive reference will be represented as `<Recursion on typename with id=number>`. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.12.2 PrettyPrinter 物件

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

建立一個*PrettyPrinter* 實例。

Arguments have the same meaning as for *pp()*. Note that they are in a different order, and that *sort_dicts* defaults to `True`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
```

(繼續下一頁)

(繼續上一頁)

```
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

在 3.4 版的變更: 新增 *compact* 參數。

在 3.8 版的變更: 新增 *sort_dicts* 參數。

在 3.10 版的變更: 新增 *underscore_numbers* 參數。

在 3.11 版的變更: No longer attempts to write to `sys.stdout` if it is `None`.

PrettyPrinter 實例有以下方法:

PrettyPrinter.**pformat**(*object*)

Return the formatted representation of *object*. This takes into account the options passed to the *PrettyPrinter* constructor.

PrettyPrinter.**pprint**(*object*)

Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new *PrettyPrinter* objects don't need to be created.

PrettyPrinter.**isreadable**(*object*)

Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using *eval()*. Note that this returns `False` for recursive objects. If the *depth* parameter of the *PrettyPrinter* is set and the object is deeper than allowed, this returns `False`.

PrettyPrinter.**isrecursive**(*object*)

Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the *saferepr()* implementation.

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)

Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the *id()* of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be `True`. Recursive calls to the *format()* method should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be `0` if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.

8.12.3 范例

To demonstrate several uses of the *pp()* function and its parameters, let's fetch information about a project from PyPI:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/1.2.0/json') as resp:
...     project_info = json.load(resp)['info']
```

In its basic form, *pp()* shows the whole object:

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
```

(繼續下一頁)

(繼續上一頁)

```
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

The result can be limited to a certain *depth* (ellipsis is used for deeper contents):

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
```

(繼續下一頁)

(繼續上一頁)

```
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Additionally, maximum character *width* can be suggested. If a long object cannot be split, the specified width will be exceeded:

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
```

(繼續下一頁)

(繼續上一頁)

```
               'New" section for the most recent version\n'
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.13 `reprlib` --- `repr()` 的替代實作

原始碼： Lib/reprlib.py

The `reprlib` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class reprlib.**Repr**(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

Class which provides formatting services useful in implementing functions similar to the built-in *repr()*; size limits for different object types are added to avoid the generation of representations which are excessively long.

The keyword arguments of the constructor can be used as a shortcut to set the attributes of the *Repr* instance. Which means that the following initialization:

```
aRepr = reprlib.Repr(maxlevel=3)
```

等價於:

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

See section *Repr Objects* for more information about *Repr* attributes.

在 3.12 版的變更: Allow attributes to be set via keyword arguments.

reprlib.**aRepr**

This is an instance of *Repr* which is used to provide the *repr()* function described below. Changing the attributes of this object will affect the size limits used by *repr()* and the Python debugger.

reprlib.**repr**(*obj*)

This is the *repr()* method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a placeholder string instead.

`@reprlib.`**`recursive_repr`**(*fillvalue='...'*)

Decorator for `__repr__()` methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual `__repr__()` call is made. For example:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

在 3.2 版被加入.

8.13.1 Repr 物件

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

`Repr.`**`fillvalue`**

This string is displayed for recursive references. It defaults to `...`.

在 3.11 版被加入.

`Repr.`**`maxlevel`**

Depth limit on the creation of recursive representations. The default is `6`.

`Repr.`**`maxdict`**

`Repr.`**`maxlist`**

`Repr.`**`maxtuple`**

`Repr.`**`maxset`**

`Repr.`**`maxfrozenset`**

`Repr.`**`maxdeque`**

`Repr.`**`maxarray`**

Limits on the number of entries represented for the named object type. The default is `4` for *maxdict*, `5` for *maxarray*, and `6` for the others.

`Repr.`**`maxlong`**

Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is `40`.

`Repr.`**`maxstring`**

Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is `30`.

`Repr.`**`maxother`**

This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is `20`.

`Repr.`**`indent`**

If this attribute is set to `None` (the default), the output is formatted with no line breaks or indentation, like the standard *repr()*. For example:

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
```

If *indent* is set to a string, each recursion level is placed on its own line, indented by that string:

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
-->-->-->3: 4.5,
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

Setting *indent* to a positive integer value behaves as if it was set to a string with that number of spaces:

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

在 3.12 版被加入.

Repr.**repr**(*obj*)

The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

Repr.**repr1**(*obj*, *level*)

Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.**repr_TYPE**(*obj*, *level*)

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.13.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # 印出 '<stdin>'
```

```
<stdin>
```

8.14 enum --- 對列舉的支援

在 3.4 版被加入.

原始碼： Lib/enum.py

Important

本頁包含 API 的參考資訊。關於教學資訊及更多進階主題的討論請參考

- 基本教學
- 進階教學
- 列舉指南

列舉：

- 是一組綁定唯一值的代表名稱（成員）
- 可以用🄵代的方式以定義的順序回傳其正式 (canonical)（即非🄵名）成員
- 使用 *call* 語法來透過值回傳成員
- 使用 *index* 語法來透過名稱回傳成員

列舉透過 `class` 語法或函式呼叫的語法來建立：

```
>>> from enum import Enum

>>> # class 語法
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # 函式語法
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

雖然我們可以用 `class` 語法來建立列舉，列舉🄵不是標準的 Python 類🄵。參考列舉有何差🄵以取得更多細節。

備🄵

命名方式

- `Color` 類🄵是一個*列舉*（或 *enum*）
- `Color.RED`、`Color.GREEN` 等屬性是*列舉成員*（或*成員*），🄵且使用上可以看作常數。
- 列舉成員有*名稱*和*值*（`Color.RED` 的名稱是 `RED`，`Color.BLUE` 的值是 `3` 諸如此類）

8.14.1 模組🄵容

EnumType

Enum 及其子類🄵的 `type`。

Enum

用來建立列舉常數的基礎類🄵。

IntEnum

用來建立列舉常數的基礎類🄵，同時也是*int* 的子類🄵。（備🄵）

StrEnum

用來建立列舉常數的基礎類🄵，同時也是*str* 的子類🄵。（備🄵）

Flag

用來建立列舉常數的基礎類🄵，可以使用位元操作來結合成員且其結果不失去*Flag* 的成員資格。

IntFlag

用來建立列舉常數的基礎類🄵，可以使用位元操作來結合成員且其結果不失去*IntFlag* 的成員資格。*IntFlag* 的成員也是*int* 的子類🄵。（備🄵）

ReprEnum

由*IntEnum*、*StrEnum* 及*IntFlag* 所使用來保留這些混合型🄵的*str()*。

EnumCheck

一個有 `CONTINUOUS`、`NAMED_FLAGS` 及 `UNIQUE` 這些值的列舉，和*verify()* 一起使用來確保給定的列舉符合多種限制。

FlagBoundary

一個有 `STRICT`、`CONFORM`、`EJECT` 及 `KEEP` 這些值的列舉，允許列舉對如何處理非法值做更細微的控制。

EnumDict

dict 的子類🄵，用於當作*EnumType* 的子類🄵時使用。

auto

列舉成員的實例會被取代成合適的值。*StrEnum* 預設是小寫版本的成員名稱，其它列舉則預設是 1 且往後遞增。

property()

允許*Enum* 成員擁有屬性且不會與成員名稱有衝突。`value` 及 `name` 屬性是用這個方式來實作。

unique()

Enum 類🄵的裝飾器，用來確保任何值只有綁定到一個名稱上。

verify()

Enum 類🄵的裝飾器，用來檢查列舉上使用者所選的限制。

member()

讓 `obj` 變成成員。可以當作裝飾器使用。

nonmember()

不讓 `obj` 變成成員。可以當作裝飾器使用。

global_enum()

修改列舉上的*str()* 及*repr()* ，讓成員顯示🄵屬於模組而不是類🄵，🄵將該列舉成員匯出到全域命名空間。

show_flag_values()

回傳旗標 (flag) 🄵包含的所有 2 的次方的整數串列。

在 3.6 版被加入: `Flag`, `IntFlag`, `auto`

在 3.11 版被加入: `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

在 3.13 版被加入: `EnumDict`

8.14.2 資料型🄵

class enum.**EnumType**

EnumType 是 *enum* 列舉的*metaclass*。*EnumType* 可以有子類🄵 -- 細節請參考 建立 EnumType 的子類🄵。

`EnumType` 負責在最後的列舉上面設定正確的 `__repr__()`、`__str__()`、`__format__()` 及 `__reduce__()` 方法，以及建立列舉成員、正確處理重🄵、提供列舉類🄵的🄵代等等。

在 3.11 版被加入: 在 3.11 之前，`EnumType` 稱作 `EnumMeta`，其目前仍可作🄵🄵名使用。

__call__(*cls*, *value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

這個方法可以用兩種不同的方式呼叫:

- 查詢已存在的成員:

  cls
  所呼叫的列舉類🄵。

  value
  要查詢的值。

- 使用 `cls` 列舉來建立新列舉（只有在現有列舉🄵有任何成員時）

  cls
  所呼叫的列舉類🄵。

  value
  要建立的新列舉的名稱。

  names
  新列舉的成員的名稱/值。

  module
  新列舉要建立在哪個模組名稱下。

qualname
這個列舉在模組🄵實際上的位置。

type
新列舉的混合型🄵。

start
列舉的第一個整數值（由*auto* 所使用）

boundary
在位元操作時怎🄵處理範圍外的值（只有*Flag* 會用到）

__contains__(*cls*, *member*)
如果 member 屬於 `cls` 則回傳 `True`：

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

在 3.12 版的變更: 在 Python 3.12 之前，如果用非列舉成員做屬於檢查 (containment check) 會引發 `TypeError`。

__dir__(*cls*)
回傳 `['__class__', '__doc__', '__members__', '__module__']` 及 *cls* 的成員名稱：

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__getitem__', '__
↪init_subclass__', '__iter__', '__len__', '__members__', '__module__', '__name__',
↪ '__qualname__']
```

__getitem__(*cls*, *name*)
回傳 *cls* 中符合 *name* 的列舉成員，或引發*KeyError*：

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)
以定義的順序回傳在 *cls* 中的每個成員：

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)
回傳 *cls* 的成員數量：

```
>>> len(Color)
3
```

__members__
回傳每個列舉名稱到其成員的對映，包括🄵名

__reversed__(*cls*)
以跟定義相反的順序回傳 *cls* 的每個成員：

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

class enum.**Enum**
Enum 是所有 *enum* 列舉的基礎類🄵。

name

用來定義 Enum 成員的名稱：

```
>>> Color.BLUE.name
'BLUE'
```

value

Enum 成員給定的值：

```
>>> Color.RED.value
1
```

成員的值，可以在*__new__()* 設定。

> **備🄵**
>
> 列舉成員的值
>
> 成員的值可以是任何值：*int*、*str* 等等。如果實際使用什🄵值🄵不重要，你可以使用*auto* 實例，它會🄵你選擇合適的值。更多細節請參考*auto*。
>
> 雖然可以使用可變的 (mutable) / 不可雜🄵的 (unhashable) 值，例如*dict*、*list* 或可變的*dataclass*，它們在建立期間會對效能🄵生相對於列舉中可變的 / 不可雜🄵的值總數量的二次方影響。

name

成員名稱。

value

成員的值，可以在*__new__()* 設定。

order

已不再使用，只🄵了向後相容而保留（類🄵屬性，在類🄵建立時移除）

ignore

ignore 只有在建立的時候用到，在列舉建立完成後會被移除。

ignore 是一個不會變成成員的名稱串列，在列舉建立完成後其名稱會被移除。範例請參考 TimePeriod。

__dir__(*self*)

回傳 `['__class__', '__doc__', '__module__', 'name', 'value']` 及任何 *self.__class__* 上定義的公開方法：

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today',
↪'value']
```

_generate_next_value_(*name*, *start*, *count*, *last_values*)

name
定義的成員名稱（例如'RED'）。

start
列舉的開始值，預設🄵 1。

count
已定義的成員數量，不包含目前這一個。

last_values
一個之前值的串列。

一個 *staticmethod*，用來🄴定`auto`下一個要回傳的值的：

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

__init__(*self*, **args*, ***kwds*)

預設情🄵下，不執行任何操作。如果在成員賦值中給出多個值，這些值將成🄴與 `__init__` 分🄴的引數；例如

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` 將被稱🄵 `Weekday.__init__(self, 1, 'Mon')`

__init_subclass__(*cls*, ***kwds*)

一個 *classmethod*，用來進一步設定後續的子類🄴，預設不做任何事。

missing(*cls*, *value*)

一個 *classmethod*，用來查詢在 *cls* 🄴找不到的值。預設不做任何事，但可以被覆寫以實作客🄵化的搜尋行🄵：

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

預設情🄵下不存在。如果有指定，無論是在列舉類🄴定義中還是在 mixin 類🄴中（例如 `int`），都將傳遞成員賦值中給出的所有值；例如

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

會🄴生呼叫 `int('1a', 16)` 而該成員的值🄴 `26`。

> **備🄵**
>
> 當寫自訂的 `__new__` 時，不要使用 `super().__new__`，而是要呼叫適當的 `__new__`。

__repr__(*self*)

回傳呼叫 *repr()* 時使用的字串。預設回傳 *Enum* 名稱、成員名稱及值，但可以被覆寫：

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

__str__(*self*)

回傳呼叫 *str()* 時使用的字串。預設回傳 *Enum* 名稱及成員名稱，但可以被覆寫：

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(*self*)

回傳呼叫 *format()* 及 *f-string* 時使用的字串。預設回傳 *__str__()* 的回傳值，但可以被覆寫：

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

> **備🄵**
>
> *Enum* 使用 *auto* 會🄴生從 `1` 開始遞增的整數。

在 3.12 版的變更: 新增 enum-dataclass-support

_add_alias_()

新增一個名稱作🄴現有成員的🄴名：

```
>>> Color.RED._add_alias_("ERROR")
>>> Color.ERROR
<Color.RED: 1>
```

如果該名稱已經被指派給不同的成員，則會引發*NameError*。

在 3.13 版被加入.

_add_value_alias_()

新增一個值作🄵現有成員的🄵名：

```
>>> Color.RED._add_value_alias_(42)
>>> Color(42)
<Color.RED: 1>
```

如果該值已經被連結到不同的成員，則會引發*ValueError*。

在 3.13 版被加入.

class enum.**IntEnum**

IntEnum 和*Enum* 一樣，但其成員同時也是整數而可以被用在任何使用整數的地方。如果 *IntEnum* 成員經過任何整數運算，結果值會失去其列舉狀態。

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

備🄵

IntEnum 使用*auto* 會🄴生從 1 開始遞增的整數。

在 3.11 版的變更: 🄵了更好地支援現存常數取代 *(replacement of existing constants)* 的使用情境，__str__() 現在會是 int.__str__()。🄵了同樣的理由，__format__() 已經是 int.__format__()。

class enum.**StrEnum**

StrEnum 和*Enum* 一樣，但其成員同時也是字串而可以被用在幾乎所有使用字串的地方。*StrEnum* 成員經過任何字串操作的結果會不再是列舉的一部份。

```
>>> from enum import StrEnum, auto
>>> class Color(StrEnum):
...     RED = 'r'
...     GREEN = 'g'
...     BLUE = 'b'
...     UNKNOWN = auto()
...
>>> Color.RED
<Color.RED: 'r'>
>>> Color.UNKNOWN
```

(繼續下一頁)

(繼續上一頁)

```
<Color.UNKNOWN: 'unknown'>
>>> str(Color.UNKNOWN)
'unknown'
```

備🄵

stdlib 🄵有些地方會檢查只能是*str* 而不是*str* 的子類🄵（也就是 `type(unknown) == str` 而不是 `isinstance(unknown, str)`），在這些地方你需要使用 `str(MyStrEnum.MY_MEMBER)`。

備🄵

StrEnum 使用*auto* 會🄵生小寫的成員名稱當作值。

備🄵

🄵了更好地支援現存常數取代 *(replacement of existing constants)* 的使用情境，`__str__()` 現在會是 `str.__str__()`。🄵了同樣的理由，`__format__()` 也會是 `str.__format__()`。

在 3.11 版被加入.

class enum.**Flag**

`Flag` 與*Enum* 相同，但其成員支援位元運算子 `&` (*AND*)、`|` (*OR*)、`^` (*XOR*) 和 `~` (*INVERT*)；這些操作的結果是列舉的成員（的🄵名）。

__contains__(*self*, *value*)

如果 value 在 self 🄵則回傳 *True*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

回傳所有包含的非🄵名成員:

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

在 3.11 版被加入.

__len__(self)：

回傳旗標🄵的成員數量：

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

在 3.11 版被加入.

__bool__(self)：

如果成員在旗標🄵則回傳 *True*，否則回傳 *False*：

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__(*self*, *other*)

回傳和 other 做 OR 過後的二進位旗標：

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__(*self*, *other*)

回傳和 other 做 AND 過後的二進位旗標：

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

__xor__(*self*, *other*)

回傳和 other 做 XOR 過後的二進位旗標：

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self)：

回傳所有在 *type(self)* 但不在 *self* 🄵的旗標：

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

用來格式化任何剩下未命名數值的函式。預設是值的 repr，常見選擇是*hex()* 和*oct()*。

備🄵

Flag 使用*auto* 會🄵生從 1 開始 2 的次方的整數。

在 3.11 版的變更: 值🄴 0 的旗標的 *repr()* 已改變。現在是:

```
>>> Color(0)
<Color: 0>
```

class enum.**IntFlag**

IntFlag 和*Flag* 一樣，但其成員同時也是整數而可以被用在任何使用整數的地方。

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

如果 *IntFlag* 成員經過任何整數運算，其結果不是 *IntFlag*:

```
>>> Color.RED + 2
3
```

如果 *IntFlag* 成員經過*Flag* 操作且:

- 結果是合法的 *IntFlag*: 回傳 *IntFlag*
- 結果不是合法的 *IntFlag*: 結果會根據*FlagBoundary* 的設定

未命名且值🄴 0 的旗標的*repr()* 已改變。現在是:

```
>>> Color(0)
<Color: 0>
```

備🄴

IntFlag 使用*auto* 會🄴生從 1 開始 2 的次方的整數。

在 3.11 版的變更: 🄴了更好地支援現存常數取代 *(replacement of existing constants)* 的使用情境，__str__() 現在會是 int.__str__()。🄴了同樣的理由，__format__() 已經是 int.__format__()。

IntFlag 的反轉 (inversion) 現在會回傳正值，該值是不在給定旗標的所有旗標聯集，而不是一個負值。這符合現有*Flag* 的行🄴。

class enum.**ReprEnum**

ReprEnum 使用*Enum* 的*repr()*，但使用混合資料型🄵的*str()*:

- 對*IntEnum* 和*IntFlag* 是 int.__str__()
- 對*StrEnum* 是 str.__str__()

繼承 ReprEnum 來保留混合資料型🄵的*str()* / *format()*，而不是使用*Enum* 預設的*str()*。

在 3.11 版被加入.

class enum.**EnumCheck**

EnumCheck 包含*verify()* 裝飾器使用的選項，以確保多樣的限制，不符合限制會🄵生*ValueError*。

UNIQUE

確保每個值只有一個名稱:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

確保在最小值成員跟最大值成員間🄵有缺少值:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

確保任何旗標群組 / 遮罩只包含命名旗標 -- 當值是用指定而不是透過*auto()* 🄵生時是很實用的:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing combined␣
↪values of 0x18 [use enum.show_flag_values(value) for details]
```

備🄵

CONTINUOUS 和 NAMED_FLAGS 是設計用來運作在整數值的成員上。

在 3.11 版被加入.

class enum.**FlagBoundary**

`FlagBoundary` 控制在*Flag* 及其子類🄵中如何處理範圍外的值。

STRICT

範圍外的值會引發*ValueError*。這是*Flag* 的預設行🄵:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

會移除範圍外的值中的非法值，留下合法的*Flag* 值:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

範圍外的值會失去它們的*Flag* 成員資格且恢復成*int*。

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

範圍外的值會被保留，*Flag* 成員資格也會被保留。這是*IntFlag* 的預設行🄵:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

在 3.11 版被加入.

class enum.**EnumDict**

EnumDict 是*dict* 的子類🄵，用來作🄴定義列舉類🄴的命名空間（參見 prepare）。它被公開來使得*EnumType* 的子類🄴能具有進階行🄵，例如讓每個成員有多個值。它應該在被呼叫時帶上正在建立的列舉類🄵名稱，否則私有名稱和🄵部類🄵將無法被正確處理。

注意只有*MutableMapping* 介面（`__setitem__()` 和*update()*）被覆寫。可能可以使用其他 `dict` 操作來繞過檢查，例如 `|=`。

member_names

一個成員名稱的串列。

在 3.13 版被加入.

支援的 __dunder__ 名稱

__members__ 是一個唯讀有序的成員名稱：成員項目的對映。只有在類🄵上可用。

__new__()，如果有指定，它必須建立🄵回傳列舉成員；適當地設定成員的 _value_ 也是一個很好的主意。一旦所有成員都建立之後就不會再被用到。

支援的 _sunder_ 名稱

- *_name_* -- 成員名稱
- *_value_* -- 成員的值；可以在 __new__ 設定
- *_missing_()* -- 當值🄵有被找到時會使用的查詢函式；可以被覆寫
- *_ignore_* -- 一個名稱的串列，可以是*list* 或*str*，它不會被轉🄵成成員，且在最後的類🄵上會被移除
- *_order_* -- 不再被使用，僅🄵了向後相容而保留（類🄵屬性，在類🄵建立時移除）
- *_generate_next_value_()* -- 用來🄵列舉成員取得合適的值；可以被覆寫

> **備🄵**
>
> 對標準的*Enum* 類🄵來🄵，下一個被選擇的值是所看過的最大值加一。
>
> 對*Flag* 類🄵來🄵，下一個被選擇的值是下一個最大的 2 的次方的數字。

- *_add_alias_()* -- 新增一個名稱作🄵現有成員的🄵名。
- *_add_value_alias_()* -- 新增一個值作🄵現有成員的🄵名。
- 雖然 _sunder_ 名稱通常保留用於*Enum* 類🄵的進一步開發而不能被使用，但有些是明確允許的：
  - _repr_*（例如 _repr_html_），例如用於 IPython 的豐富顯示

在 3.6 版被加入: _missing_、_order_、_generate_next_value_

在 3.7 版被加入: _ignore_

在 3.13 版被加入: _add_alias_、_add_value_alias_、_repr_*

8.14.3 通用項目與裝飾器

class enum.**auto**

auto 可以用來取代給值。如果使用的話，*Enum* 系統會呼叫*Enum* 的*_generate_next_value_()* 來取得合適的值。對*Enum* 和*IntEnum* 來🄵，合適的值是最後一個值加一；對*Flag* 和*IntFlag* 來🄵，是第一個比最大值還大的 2 的次方的數字；對*StrEnum* 來🄵，是成員名稱的小寫版本。如果混用 *auto()* 和手動指定值的話要特🄵注意。

auto 實例只有在最上層的賦值時才會被解析：

- FIRST = auto() 可以運作（auto() 會被取代成 1）
- SECOND = auto(), -2 可以運作（auto 會被取代成 2, 因此 2, -2 會被用來建立列舉成員 SECOND;
- THREE = [auto(), -3] 無法運作（<auto 實例>, -3 會被用來建立列舉成員 THREE）

在 3.11.1 版的變更: 在之前的版本中，auto() 必須是賦值行🄵的唯一🄵容才能運作正確。

可以覆寫 _generate_next_value_ 來客🄵 *auto* 使用的值。

> **備🄵**
>
> 在 3.13 預設 _generate_next_value_ 總是回傳最大的成員值加一，如果任何成員是不相容的型🄵就會失敗。

`@enum.property`

和🄵建的 *property* 相似的裝飾器，但只專門針對列舉。它允許成員屬性和成員本身有相同名稱。

> **備🄵**
>
> 屬性和成員必須定義在分開的類🄵🄵；例如 *value* 和 *name* 屬性定義在 *Enum* 類🄵而 *Enum* 子類🄵可以定義成員名稱🄵 `value` 和 `name`。

在 3.11 版被加入.

`@enum.unique`

專門針對列舉的 `class` 裝飾器。它搜尋列舉的*__members__*，🄵集任何它找到的🄵名；如果有找到任何🄵名則引發*ValueError* 🄵附上細節：

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

`@enum.verify`

專門針對列舉的 `class` 裝飾器。使用*EnumCheck* 🄵的成員來指定在裝飾的列舉上應該檢查什🄵限制。

在 3.11 版被加入.

`@enum.member`

列舉所使用的裝飾器：其目標會變成成員。

在 3.11 版被加入.

`@enum.nonmember`

列舉所使用的裝飾器：其目標不會變成成員。

在 3.11 版被加入.

`@enum.global_enum`

修改列舉的*str()* 及*repr()* 的裝飾器，讓成員顯示🄵屬於模組而不是其類🄵。應該只有當列舉成員被匯出到模組的全域命名空間才使用（範例請參考*re.RegexFlag*）。

在 3.11 版被加入.

`enum.show_flag_values`(*value*)

回傳在旗標值中包含的所有 2 的次方的整數串列。

在 3.11 版被加入.

8.14.4 備🄵

IntEnum、*StrEnum* 及 *IntFlag*

這三種列舉型🄵是設計來直接取代現有以整數及字串🄵基底的值；因此它們有額外的限制：

- `__str__` 使用值而不是列舉成員的名稱
- `__format__` 因🄵使用 `__str__`，也會使用值而不是列舉成員的名稱

如果你不需要或不想要這些限制，你可以透過混合 `int` 或 `str` 型🄵來建立自己的基礎類🄵：

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

或者你也可以在你的列舉重新給定合適的*str()*：

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

8.15 `graphlib` —-- 使用類圖 (graph-like) 結構進行操作的功能

原始碼： Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

提供對包含可雜🄵 *(hashable)* 節點之圖 (graph) 進行拓撲排序 (topologically sort) 的功能。

拓撲排序是圖中頂點 (vertex) 的🄵性排序，使得對於從頂點 u 到頂點 v 的每條有向邊 (directed edge) u -> v，頂點 u 在排序中會位於頂點 v 之前。例如，圖的頂點可能代表要執行的任務，而邊可能代表一個任務必須在另一個任務之前執行的限制；在此範例中，拓撲排序只是任務的一種有效序列。若且唯若 (if and only if) 圖🄵有有向環 (directed cycle) 時，即如果它是個有向無環圖 (directed acyclic graph)，則完整的拓撲排序才是可行的。

如果提供了可選的 *graph* 引數，它必須是表示有向無環圖的字典，其中鍵是節點，值是圖中該節點的包含所有前驅節點 (predecessor) 之可🄵代物件（這些前驅節點具有指向以鍵表示之節點的邊）。可以使用*add()* 方法將其他節點新增到圖中。

在一般情🄵下，對給定的圖執行排序所需的步驟如下：

- 以選用的初始圖建立*TopologicalSorter* 的實例。
- 在圖中新增其他節點。
- 呼叫圖的*prepare()*。
- 當*is_active()* 🄵 `True` 時，🄵代*get_ready()* 回傳的節點🄵處理它們。在每個節點完成處理時呼叫*done()*。

如果只需要立即對圖中的節點進行排序且不涉及平行性 (parallelism)，則可以直接使用便捷方法*TopologicalSorter.static_order()*：

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

該類🄵設計🄵在節點准備就緒時，簡單支援節點的平行處理。例如：

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on off the
        # 'task_queue' queue.
        task_queue.put(node)

    # When the work for a node is done, workers put the node in
    # 'finalized_tasks_queue' so we can get more nodes to work on.
    # The definition of 'is_active()' guarantees that, at this point, at
    # least one node has been placed on 'task_queue' that hasn't yet
    # been passed to 'done()', so this blocking 'get()' must (eventually)
    # succeed.  After calling 'done()', we loop back to call 'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as soon as
    # logically possible.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

向圖中新增新節點及其前驅節點。*node* 和 *predecessors* 中的所有元素都必須是可雜🄵的。

如果以相同節點引數多次呼叫，則依賴項的集合將會是傳入的所有依賴項的聯集。

可以新增一個🄵有依賴關🄵的節點（*predecessors* 未提供）或提供兩次依賴關🄵。如果有之前未曾提供的節點被包含在 *predecessors* 中，它將自動新增到🄵有前驅節點的圖中。

如果在*prepare()* 之後呼叫，則引發*ValueError*。

prepare()

將圖標記🄵已完成🄵檢查圖中的循環。如果檢測到任何循環，將引發*CycleError*，但*get_ready()* 仍可用於盡可能獲得更多的節點，直到循環阻塞了進度。呼叫此函式後就無法修改圖，因此無法使用*add()* 來新增更多節點。

A *ValueError* will be raised if the sort has been started by *static_order()* or *get_ready()*.

在 3.14 版的變更: `prepare()` can now be called more than once as long as the sort has not started. Previously this raised *ValueError*.

is_active()

如果可以有更多進度則回傳 `True`，否則回傳 `False`。如果循環不阻塞解析 (resolution) 🄵且仍有節點准備就緒但尚未由*TopologicalSorter.get_ready()* 回傳或標記🄵*TopologicalSorter.done()* 的節點數量較*TopologicalSorter.get_ready()* 所回傳的少，則可以繼續取得進度。

此類🄵的 `__bool__()` 方法遵循此函式，因此以下做法：

```
if ts.is_active():
    ...
```

可以簡單地用以下方式替🄵：

```
if ts:
    ...
```

如果呼叫之前🄵有先呼叫*prepare()* 則引發*ValueError*。

done(**nodes*)

將*TopologicalSorter.get_ready()* 回傳的一組節點標記🄵已處理，停止阻塞 *nodes* 中每個節點的任何後繼節點 (successor)，以便將來通過呼叫*TopologicalSorter.get_ready()* 回傳。

若🄵有和該呼叫一起呼叫*prepare()* 或節點還🄵有被*get_ready()* 回傳，且如果 *nodes* 中有任何節點已被先前對此方法的呼叫標記🄵已處理、或者未使用*TopologicalSorter.add()* 將節點新增到圖中，則引發*ValueError*。

get_ready()

回傳一個包含所有准備就緒節點的 `tuple`。最初它回傳🄵有前驅節點的所有節點，一旦通過呼叫*TopologicalSorter.done()* 來將這些節點標記🄵已處理後，進一步的呼叫將回傳所有其全部前驅節點都已被處理的新節點。若無法取得更多進度，將回傳空 tuple。

如果呼叫之前🄵有先呼叫*prepare()* 則引發*ValueError*。

static_order()

回傳一個可🄵代物件，它將按拓撲排序🄵代節點。使用此方法時，不應呼叫*prepare()* 和*done()*。此方法等效於：

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

回傳的特定順序可能取🄵於將項目插入圖中的特定順序。例如：

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

這是因🄵"0" 和"2" 在圖中處於同一級🄵（它們將在對*get_ready()* 的同一呼叫中回傳）🄵且它們之間的順序取🄵於插入順序。

如果檢測到任何循環，則引發*CycleError*。

在 3.9 版被加入.

8.15.1 例外

graphlib 模組定義了以下例外類🄵：

exception graphlib.**CycleError**

ValueError 的子類🄵，如果作用的圖中存在循環則由*TopologicalSorter.prepare()* 引發。如果存在多個循環，則只會報告未定義的其中一個🄵包含在例外中。

檢測到的循環可以通過例外實例的*args* 屬性中第二個元素來存取，其🄵一個節點列表，每個節點在圖中都是列表中下一個節點的直接前驅節點（immediate predecessor，即父節點）。在報告列表中，第一個和最後一個節點將會是相同的，用以明確表示它是循環的。

CHAPTER 9

數值與數學模組

本章所描述的模組提供了數值和與數學相關的函式和資料型🄵。*numbers* 模組定義了數值型🄵的抽象階層結構。*math* 和 *cmath* 模組包含了用於浮點數和🄵數的各種數學函式。*decimal* 模組支援對十進位數字的精確表示以及任意精度的算術運算。

本章節包含下列的模組：

9.1 numbers --- 數值的抽象基底類🄵

原始碼： Lib/numbers.py

numbers 模組 (**PEP 3141**) 定義了數值抽象基底類🄵 的階層結構，其中逐一定義了更多操作。此模組中定義的型🄵都不可被實例化。

class numbers.**Number**

數值階層結構的基礎。如果你只想確認引數 *x* 是不是數值、🄵不關心其型🄵，請使用 `isinstance(x, Number)`。

9.1.1 數值的階層

class numbers.**Complex**

這個型🄵的子類🄵描述了🄵數🄵包含適用於🄵建 *complex* 型🄵的操作。這些操作有：*complex* 和 *bool* 的轉🄵、*real*、*imag*、+、-、*、/、**、*abs()*、*conjugate()*、== 以及 !=。除 - 和 != 之外所有操作都是抽象的。

real

🄵抽象的。取得該數值的實數部分。

imag

🄵抽象的。取得該數值的🄵數部分。

abstractmethod conjugate()

🄵抽象的。回傳共🄵🄵數，例如 `(1+3j).conjugate() == (1-3j)`。

class numbers.**Real**

相對於*Complex*，Real 加入了只有實數才能進行的操作。

簡單的🄵，有*float* 的轉🄵、*math.trunc()*、*round()*、*math.floor()*、*math.ceil()*、*divmod()*、//、%、<、<=、>、和 >=。

實數同樣提供*complex()*、*real*、*imag* 和*conjugate()* 的預設值。

class numbers.**Rational**

Real 的子型🄵，🄵增加了*numerator* 和*denominator* 這兩種特性。它也會提供*float()* 的預設值。

numerator 和*denominator* 的值必須是*Integral* 的實例且*denominator* 要是正數。

numerator

🄵抽象的。

denominator

🄵抽象的。

class numbers.**Integral**

Rational 的子型🄵，🄵增加了*int* 的轉🄵操作。🄵*float()*、*numerator* 和*denominator* 提供了預設值。🄵*pow()* 方法增加了求余 (modulus) 和位元字串運算 (bit-string operations) 的抽象方法：<<、>>、&、^、|、~。

9.1.2 給型🄵實作者的🄵記

實作者需注意，相等的數值除了大小相等外，還必須擁有同樣的雜🄵值。當使用兩個不同的實數擴充時，這可能是很微妙的。例如，*fractions.Fraction* 底下的*hash()* 實作如下：

```
def __hash__(self):
    if self.denominator == 1:
        # 正確處理整數。
        return hash(self.numerator)
    # 檢查費時，但🄵對正確。
    if self == float(self):
        return hash(float(self))
    else:
        # 使用 tuple 的雜🄵值以避免簡單分數上的高碰撞率。
        return hash((self.numerator, self.denominator))
```

加入更多數值 ABC

當然，還有更多用於數值的 ABC，如果不加入它們就不會有健全的階層。你可以在*Complex* 和*Real* 中加入 MyFoo，像是：

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

實作算術操作

我們想要實作算術操作，來使得混合模式操作要🄵呼叫一個作者知道兩個引數之型🄵的實作，要🄵將其轉🄵成最接近的🄵建型🄵🄵執行這個操作。對於*Integral* 的子型🄵，這意味著 __add__() 和 __radd__() 必須用如下方式定義：

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
```

(繼續下一頁)

(繼續上一頁)

```
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Complex 的子類🄵有 5 種不同的混合型🄵操作。我將上面提到所有不涉及 `MyIntegral` 和 `OtherTypeIKnowAbout` 的程式碼稱作「模板 (boilerplate)」。`a` 是*Complex* 之子型🄵 `A` 的實例 (`a : A <: Complex`)，同時 `b : B <: Complex`。我將要計算 `a + b`：

1. 如果 `A` 有定義成一個接受 `b` 的 `__add__()`，不會發生問題。
2. 如果 `A` 回退成模板程式碼，它將回傳一個來自 `__add__()` 的值，🄵喪失讓 `B` 定義一個更完善的 `__radd__()` 的機會，因此模板需要回傳一個來自 `__add__()` 的*NotImplemented*。(或者 `A` 可能完全不實作 `__add__()`。)
3. 接著看 `B` 的 `__radd__()`。如果它接受 `a`，不會發生問題。
4. 如果🄵有成功回退到模板，就🄵有更多的方法可以去嘗試，因此這🄵將使用預設的實作。
5. 如果 `B <: A`，Python 會在 `A.__add__` 之前嘗試 `B.__radd__`。這是可行的，因🄵它是透過對 `A` 的理解而實作的，所以這可以在交給*Complex* 之前處理好這些實例。

如果 `A <: Complex` 和 `B <: Real` 且🄵有共享任何其他型🄵上的理解，那🄵適當的共享操作會涉及🄵建的*complex*，🄵且分🄵用到 `__radd__()`，因此 `a+b == b+a`。

由於大部分對任意給定類型的操作都十分相似的，定義一個🄵任意給定運算子生成向前 (forward) 與向後 (reverse) 實例的輔助函式可能會非常有用。例如，*fractions.Fraction* 使用了：

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # 包含整數。
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
```

(繼續下一頁)

(繼續上一頁)

```
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 `math` --- 數學函式

此模組提供對常見數學函式與常數的存取，包括於 C 標準中定義的那些。

這些函式不支援🄵數；若你需要計算🄵數，請使用 *cmath* 模組的同名函式。這是因🄵大多數的使用者🄵不想學習那🄵多理解🄵數所需的數學概念，所以根據支援🄵數與否分🄵兩種函式。收到一個例外而非🄵數回傳值，有助於程式設計師提早察覺參數中包含非預期的🄵數，進而從源頭查出導致此情🄵的原因。

此模組提供下列函式。除非特意🄵明，否則回傳值皆🄵浮點數。

數論函式	
comb(n, k)	從 *n* 個物品中不重🄵且不考慮排序地取出 *k* 個物品的方法數。
factorial(n)	*n* factorial
*gcd(*integers)*	Greatest common divisor of the integer arguments
isqrt(n)	Integer square root of a nonnegative integer *n*
*lcm(*integers)*	Least common multiple of the integer arguments
perm(n, k)	從 *n* 個物品中不重🄵但考慮排序地取出 *k* 個物品的方法數。
Floating point arithmetic	
ceil(x)	Ceiling of *x*, the smallest integer greater than or equal to *x*
fabs(x)	*x* 的🄵對值。
floor(x)	Floor of *x*, the largest integer less than or equal to *x*
fma(x, y, z)	Fused multiply-add operation: `(x * y) + z`
fmod(x, y)	Remainder of division `x / y`
modf(x)	Fractional and integer parts of *x*
remainder(x, y)	Remainder of *x* with respect to *y*
trunc(x)	Integer part of *x*
浮點數操作函式	
copysign(x, y)	Magnitude (absolute value) of *x* with the sign of *y*
frexp(x)	Mantissa and exponent of *x*
isclose(a, b, rel_tol, abs_tol)	Check if the values *a* and *b* are close to each other
isfinite(x)	Check if *x* is neither an infinity nor a NaN
isinf(x)	Check if *x* is a positive or negative infinity
isnan(x)	Check if *x* is a NaN (not a number)
ldexp(x, i)	`x * (2**i)`, inverse of function *frexp()*
nextafter(x, y, steps)	Floating-point value *steps* steps after *x* towards *y*
ulp(x)	Value of the least significant bit of *x*
Power, exponential and logarithmic functions	
cbrt(x)	Cube root of *x*
exp(x)	*e* raised to the power *x*

繼續

表格 1 – 繼續上一頁

exp2(x)	*2* raised to the power *x*
expm1(x)	*e* raised to the power *x*, minus 1
log(x, base)	Logarithm of *x* to the given base (*e* by default)
log1p(x)	Natural logarithm of *1+x* (base *e*)
log2(x)	Base-2 logarithm of *x*
log10(x)	Base-10 logarithm of *x*
pow(x, y)	*x* raised to the power *y*
sqrt(x)	*x* 的平方根
Summation and product functions	
dist(p, q)	Euclidean distance between two points *p* and *q* given as an iterable of coor
fsum(iterable)	Sum of values in the input *iterable*
*hypot(*coordinates)*	Euclidean norm of an iterable of coordinates
prod(iterable, start)	Product of elements in the input *iterable* with a *start* value
sumprod(p, q)	Sum of products from two iterables *p* and *q*
Angular conversion	
degrees(x)	Convert angle *x* from radians to degrees
radians(x)	Convert angle *x* from degrees to radians
Trigonometric functions	
acos(x)	Arc cosine of *x*
asin(x)	Arc sine of *x*
atan(x)	Arc tangent of *x*
atan2(y, x)	`atan(y / x)`
cos(x)	Cosine of *x*
sin(x)	Sine of *x*
tan(x)	Tangent of *x*
Hyperbolic functions	
acosh(x)	Inverse hyperbolic cosine of *x*
asinh(x)	Inverse hyperbolic sine of *x*
atanh(x)	Inverse hyperbolic tangent of *x*
cosh(x)	Hyperbolic cosine of *x*
sinh(x)	Hyperbolic sine of *x*
tanh(x)	Hyperbolic tangent of *x*
Special functions	
erf(x)	Error function at *x*
erfc(x)	Complementary error function at *x*
gamma(x)	Gamma function at *x*
lgamma(x)	Natural logarithm of the absolute value of the Gamma function at *x*
常數	
pi	$\pi = 3.141592...$
e	$e = 2.718281...$
tau	$\tau = 2\pi = 6.283185...$
inf	正無限大
nan	”Not a number” (NaN)

9.2.1 數論函式

`math.`**`comb`**`(`*n*, *k*`)`

回傳從 *n* 個物品中不重🄵且不考慮排序地取出 *k* 個物品的方法數。

當 `k <= n` 時其值🄵 `n! / (k! * (n - k)!)`，否則其值🄵 `0`。

因🄵此值等同於 `(1 + x)**n` 進行多項式展開後第 k 項的🄵數，所以又稱🄵二項式🄵數。

當任一參數非整數型🄵時會引發 *TypeError*。當任一參數🄵負數時會引發 *ValueError*。

在 3.8 版被加入.

`math.`**`factorial`**`(`*n*`)`

Return factorial of the nonnegative integer *n*.

在 3.10 版的變更: 其值🄴整數的浮點數（如：`5.0`）已不再被接受。

`math.`**`gcd`**(**integers*)

回傳指定整數引數的最大公因數。若存在任一非零引數，回傳值🄵所有引數共有因數中最大的正整數。若所有引數皆🄵零，則回傳值🄵 `0`。`gcd()` 若未傳入任何引數也將回傳 `0`。

在 3.5 版被加入.

在 3.9 版的變更: 新增支援任意數量的引數。先前僅支援兩個引數。

`math.`**`isqrt`**(*n*)

回傳非負整數 *n* 的整數平方根。此值🄵 *n* 精確平方根經下取整的值，亦等同於滿足 $a^2 \leq n$ 的最大整數值 *a*。

於有些應用程式中，取得滿足 $n \leq a^2$ 的最小整數值 *a* —— 或者🄵 *n* 精確平方根經上取整的值 —— 會更加方便。對正數 *n*，此值可使用 `a = 1 + isqrt(n - 1)` 計算。

在 3.8 版被加入.

`math.`**`lcm`**(**integers*)

回傳指定整數引數的最小公倍數。若所有引數值皆非零，回傳值🄵所有引數共有倍數中最小的正整數。若存在任一引數值🄵零，則回傳值🄵 `0`。`lcm()` 若未傳入任何引數將回傳 `1`。

在 3.9 版被加入.

`math.`**`perm`**(*n*, *k=None*)

Return the number of ways to choose *k* items from *n* items without repetition and with order.

Evaluates to `n! / (n - k)!` when `k <= n` and evaluates to zero when `k > n`.

If *k* is not specified or is `None`, then *k* defaults to *n* and the function returns `n!`.

當任一參數非整數型🄵時會引發*TypeError*。當任一參數🄵負數時會引發*ValueError*。

在 3.8 版被加入.

9.2.2 Floating point arithmetic

`math.`**`ceil`**(*x*)

回傳 *x* 經上取整的值，即大於或等於 *x* 的最小整數。若 *x* 🄵非浮點數，此函式將委派給 `x.__ceil__`，🄵回傳*Integral* 型🄵的值。

`math.`**`fabs`**(*x*)

回傳 *x* 的🄵對值。

`math.`**`floor`**(*x*)

回傳 *x* 經下取整的值，即小於或等於 *x* 的最大整數。若 *x* 🄵非浮點數，此函式將委派給 `x.__floor__`，🄵回傳*Integral* 型🄵的值。

`math.`**`fma`**(*x*, *y*, *z*)

Fused multiply-add operation. Return `(x * y) + z`, computed as though with infinite precision and range followed by a single round to the `float` format. This operation often provides better accuracy than the direct expression `(x * y) + z`.

This function follows the specification of the fusedMultiplyAdd operation described in the IEEE 754 standard. The standard leaves one case implementation-defined, namely the result of `fma(0, inf, nan)` and `fma(inf, 0, nan)`. In these cases, `math.fma` returns a NaN, and does not raise any exception.

在 3.13 版被加入.

`math.`**`fmod`**(*x*, *y*)

回傳 `x / y` 的浮點數余數，其以平臺上的 C 函式庫 `fmod(x, y)` 函式定義。請注意此函式與 Python 運算式 `x % y` 可能不會回傳相同結果。C 標準要求 `fmod(x, y)` 的回傳值完全等同（數學定義上，即無限精度）於 `x - n*y`，*n* 🄵可使回傳值與 *x* 同號且長度小於 `abs(y)` 的整數。Python 運算式 `x % y` 的回傳值則與 *y* 同號，且可能無法精確地計算浮點數引數。例如：`fmod(-1e-100, 1e100)` 的

值🄵 `-1e-100`，但 Python 運算式 `-1e-100 % 1e100` 的結果🄵 `1e100-1e-100`，此值無法准確地表示成浮點數，🄵會四舍五入🄵出乎意料的 `1e100`。因此，處理浮點數時通常會選擇函式*`fmod()`*，而處理整數時會選擇 Python 運算式 `x % y`。

`math.`**`modf`**(*x*)

Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

Note that *`modf()`* has a different call/return pattern than its C equivalents: it takes a single argument and return a pair of values, rather than returning its second return value through an 'output parameter' (there is no such thing in Python).

`math.`**`remainder`**(*x*, *y*)

Return the IEEE 754-style remainder of *x* with respect to *y*. For finite *x* and finite nonzero *y*, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise *`ValueError`* for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating point, the result of this operation is always exactly representable: no rounding error is introduced.

在 3.7 版被加入.

`math.`**`trunc`**(*x*)

Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to *`floor()`* for positive *x*, and equivalent to *`ceil()`* for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an *`Integral`* value.

For the *`ceil()`*, *`floor()`*, and *`modf()`* functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

9.2.3 Floating point manipulation functions

`math.`**`copysign`**(*x*, *y*)

回傳與 *x* 相同長度（🄵對值）且與 *y* 同號的浮點數。在支援帶符號零的平臺上，`copysign(1.0, -0.0)` 回傳 *-1.0*。

`math.`**`frexp`**(*x*)

以 `(m, e)` 對的格式回傳 *x* 的尾數 *m* 及指數 *e*。*m* 是浮點數而 *e* 是整數，且兩者精確地使 `x == m * 2**e`。若 *x* 🄵零，回傳 `(0.0, 0)`，否則令 `0.5 <= abs(m) < 1`。此函式用於以可🄵的方式「分割」浮點數🄵部表示法。

Note that *`frexp()`* has a different call/return pattern than its C equivalents: it takes a single argument and return a pair of values, rather than returning its second return value through an 'output parameter' (there is no such thing in Python).

`math.`**`isclose`**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

若 *a* 及 *b* 兩值足🄵接近便回傳 `True`，否則回傳 `False`。

兩數是否足🄵接近取🄵於給定的🄵對及相對容許偏差 (tolerance)。如果🄵有錯誤發生，結果將🄵: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

rel_tol 🄵相對容許偏差 —— *a* 與 *b* 兩數差的最大容許值，與 *a* 及 *b* 兩數的🄵對值中較大者相關。例如欲設置 5% 的容許偏差，則傳入 `rel_tol=0.05`。其預設值🄵 `1e-09`，該值可確保兩數於大約 9 個十進數位🄵相同。*rel_tol* 須不🄵負且小於 `1.0`。

abs_tol is the absolute tolerance; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol * abs(x)`, which is `False` for any nonzero `x` and *rel_tol* less than `1.0`. So add an appropriate positive *abs_tol* argument to the call.

定義於 IEEE 754 浮點標准中的特殊值 `NaN`、`inf` 和 `-inf` 會根據該標准處理。更明確地🄵，`NaN` 不會與包含自身在🄵的任何數字足🄵接近，而 `inf` 及 `-inf` 皆只與自身接近。

在 3.5 版被加入.

也參考

PEP 485 —— 用於測試近似相等的函式

`math.`**`isfinite`**(*x*)

若 *x* 不是無限值或 `NaN` 便回傳 `True`，否則回傳 `False`。（注意 `0.0` 被視🄵有限數。）

在 3.2 版被加入.

`math.`**`isinf`**(*x*)

若 *x* 是正無限值或負無限值便回傳 `True`，否則回傳 `False`。

`math.`**`isnan`**(*x*)

若 *x* 是 `NaN` —— 即非數字（NaN, not a number）—— 便回傳 `True`，否則回傳 `False`。

`math.`**`ldexp`**(*x*, *i*)

回傳 `x * (2**i)`。此函式本質上🄵*`frexp()`* 的反函式。

`math.`**`nextafter`**(*x*, *y*, *steps=1*)

Return the floating-point value *steps* steps after *x* towards *y*.

If *x* is equal to *y*, return *y*, unless *steps* is zero.

範例：

- `math.nextafter(x, math.inf)` goes up: towards positive infinity.
- `math.nextafter(x, -math.inf)` goes down: towards minus infinity.
- `math.nextafter(x, 0.0)` goes towards zero.
- `math.nextafter(x, math.copysign(math.inf, x))` goes away from zero.

另請參🄵*`math.ulp()`*。

在 3.9 版被加入.

在 3.12 版的變更: 新增 *steps* 引數。

`math.`**`ulp`**(*x*)

Return the value of the least significant bit of the float *x*:

- If *x* is a NaN (not a number), return *x*.
- 若 *x* 🄵負值，回傳 `ulp(-x)`。
- 若 *x* 🄵正無限值，回傳 *x*。
- If *x* is equal to zero, return the smallest positive *denormalized* representable float (smaller than the minimum positive *normalized* float, *`sys.float_info.min`*).
- If *x* is equal to the largest positive representable float, return the value of the least significant bit of *x*, such that the first float smaller than *x* is `x - ulp(x)`.
- Otherwise (*x* is a positive finite number), return the value of the least significant bit of *x*, such that the first float bigger than *x* is `x + ulp(x)`.

ULP stands for "Unit in the Last Place".

See also *`math.nextafter()`* and *`sys.float_info.epsilon`*.

在 3.9 版被加入.

9.2.4 Power, exponential and logarithmic functions

math.**cbrt**(*x*)

Return the cube root of *x*.

在 3.11 版被加入.

math.**exp**(*x*)

Return *e* raised to the power *x*, where *e* = 2.718281... is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

math.**exp2**(*x*)

Return *2* raised to the power *x*.

在 3.11 版被加入.

math.**expm1**(*x*)

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the *expm1()* function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

在 3.2 版被加入.

math.**log**(*x*[, *base*])

With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

math.**log1p**(*x*)

Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.**log2**(*x*)

Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

在 3.3 版被加入.

> **也參考**
>
> *int.bit_length()* returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.**log10**(*x*)

Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

math.**pow**(*x*, *y*)

Return *x* raised to the power *y*. Exceptional cases follow the IEEE 754 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when *x* is a zero or a NaN. If both *x* and *y* are finite, *x* is negative, and *y* is not an integer then `pow(x, y)` is undefined, and raises *ValueError*.

Unlike the built-in `**` operator, *math.pow()* converts both its arguments to type *float*. Use `**` or the built-in *pow()* function for computing exact integer powers.

在 3.11 版的變更: The special cases `pow(0.0, -inf)` and `pow(-0.0, -inf)` were changed to return `inf` instead of raising *ValueError*, for consistency with IEEE 754.

math.**sqrt**(*x*)

Return the square root of *x*.

9.2.5 Summation and product functions

math.**dist**(*p*, *q*)

Return the Euclidean distance between two points *p* and *q*, each given as a sequence (or iterable) of coordinates. The two points must have the same dimension.

Roughly equivalent to:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

在 3.8 版被加入.

math.**fsum**(*iterable*)

回傳可🄵代物件（iterable）中所有值的精確浮點數和。透過追🄵過程中多個部分和（partial sum）以避免精確度損失。

此演算法准確性奠基於保證 IEEE-754 浮點標准及典型奇進偶舍（half-even）模式。於有些非 Windows 平台建置時，底層 C 函式庫使用延伸精度加法運算，而可能導致對過程中同一部分和重🄵舍入，🄵使其最低有效位不如預期。

更深入的討論及兩種替代做法請參🄵 ASPN cookbook recipes 精准的浮點數總和。

math.**hypot**(**coordinates*)

Return the Euclidean norm, `sqrt(sum(x**2 for x in coordinates))`. This is the length of the vector from the origin to the point given by the coordinates.

For a two dimensional point `(x, y)`, this is equivalent to computing the hypotenuse of a right triangle using the Pythagorean theorem, `sqrt(x*x + y*y)`.

在 3.8 版的變更: Added support for n-dimensional points. Formerly, only the two dimensional case was supported.

在 3.10 版的變更: Improved the algorithm's accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

math.**prod**(*iterable*, *, *start=1*)

Calculate the product of all the elements in the input *iterable*. The default *start* value for the product is `1`.

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

在 3.8 版被加入.

math.**sumprod**(*p*, *q*)

Return the sum of products of values from two iterables *p* and *q*.

Raises *ValueError* if the inputs do not have the same length.

Roughly equivalent to:

```
sum(map(operator.mul, p, q, strict=True))
```

For float and mixed int/float inputs, the intermediate products and sums are computed with extended precision.

在 3.12 版被加入.

9.2.6 Angular conversion

math.**degrees**(*x*)

Convert angle *x* from radians to degrees.

math.**radians**(*x*)

Convert angle *x* from degrees to radians.

9.2.7 Trigonometric functions

math.**acos**(*x*)

Return the arc cosine of *x*, in radians. The result is between `0` and `pi`.

math.**asin**(*x*)

Return the arc sine of *x*, in radians. The result is between `-pi/2` and `pi/2`.

math.**atan**(*x*)

Return the arc tangent of *x*, in radians. The result is between `-pi/2` and `pi/2`.

math.**atan2**(*y*, *x*)

Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of *atan2()* is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

math.**cos**(*x*)

Return the cosine of *x* radians.

math.**sin**(*x*)

Return the sine of *x* radians.

math.**tan**(*x*)

Return the tangent of *x* radians.

9.2.8 Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

math.**acosh**(*x*)

Return the inverse hyperbolic cosine of *x*.

math.**asinh**(*x*)

Return the inverse hyperbolic sine of *x*.

math.**atanh**(*x*)

Return the inverse hyperbolic tangent of *x*.

math.**cosh**(*x*)

Return the hyperbolic cosine of *x*.

math.**sinh**(*x*)

Return the hyperbolic sine of *x*.

math.**tanh**(*x*)

Return the hyperbolic tangent of *x*.

9.2.9 特殊函式

math.**erf**(*x*)

Return the error function at *x*.

The *erf()* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

在 3.2 版被加入.

math.**erfc**(*x*)

Return the complementary error function at *x*. The complementary error function is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance.

在 3.2 版被加入.

math.**gamma**(*x*)

Return the Gamma function at *x*.

在 3.2 版被加入.

math.**lgamma**(*x*)

Return the natural logarithm of the absolute value of the Gamma function at *x*.

在 3.2 版被加入.

9.2.10 常數

math.**pi**

The mathematical constant $\pi = 3.141592...$, to available precision.

math.**e**

The mathematical constant $e = 2.718281...$, to available precision.

math.**tau**

The mathematical constant $\tau = 6.283185...$, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

在 3.6 版被加入.

math.**inf**

A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.

在 3.5 版被加入.

math.**nan**

A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *`isnan()`* function to test for NaNs instead of `is` or `==`. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

在 3.5 版被加入.

在 3.11 版的變更: It is now always available.

CPython 實作細節： The *`math`* module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise *`ValueError`* for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and *`OverflowError`* for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of

the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

也參考

cmath **模組**
Complex number versions of many of these functions.

9.3 cmath --- 🄵數的數學函式

本模組提供一些適用於🄵數的數學函式。本模組中的函式接受整數、浮點數或🄵數作🄵引數。它們也接受任何具有 `__complex__()` 或 `__float__()` 方法的 Python 物件：這些方法分🄵用於將物件轉🄵🄵🄵數或浮點數，然後再將函式應用於轉🄵後的結果。

備🄵

對於涉及分枝切割 (branch cut) 的函式，我們面臨的問題是🄵定如何定義在切割本身上的這些函式。遵循 Kahan 的論文"Branch cuts for complex elementary functions"，以及 C99 的附🄵 G 和後來的 C 標准，我們使用零符號來區分分枝切割的兩側：對於沿著（一部分）實數軸的分枝切割，我們查看🄵部的符號，而對於沿🄵軸的分枝切割，我們則查看實部的符號。

例如*cmath.sqrt()* 函式具有一條沿負實軸的分枝切割。引數 `-2-0j` 被視🄵位於分枝切割 下方處理，因此給出的結果在負🄵軸上：

```
>>> cmath.sqrt(-2-0j)
-1.4142135623730951j
```

但是引數 `-2+0j` 會被當成位於分枝切割上方處理：

```
>>> cmath.sqrt(-2+0j)
1.4142135623730951j
```

轉🄵到極座標和從極座標做轉🄵	
phase(z)	回傳 *z* 的相位角。
polar(z)	回傳 *z* 在極座標中的表達方式。
rect(r, phi)	透過極座標 *r* 和 *phi* 回傳🄵數 *z*。
🄵函數和對數函數	
exp(z)	回傳 *e* 的 *z* 次方。
log(z[, base])	回傳 *z* 以給定 *base* 🄵底（預設🄵 *e*）的對數。
log10(z)	回傳 *z* 以 10 🄵底的對數。
sqrt(z)	回傳 *z* 的平方根。
三角函數	
acos(z)	回傳 *z* 的反余弦值。
asin(z)	回傳 *z* 的反正弦值。
atan(z)	回傳 *z* 的反正切值。
cos(z)	回傳 *z* 的余弦值。
sin(z)	回傳 *z* 的正弦值。
tan(z)	回傳 *z* 的正切值。
雙曲函數	

繼續下一頁

表格 2 – 繼續上一頁

`acosh(z)`	回傳 *z* 的反雙曲余弦值。
`asinh(z)`	回傳 *z* 的反雙曲正弦值。
`atanh(z)`	回傳 *z* 的反雙曲正切值。
`cosh(z)`	回傳 *z* 的雙曲余弦值。
`sinh(z)`	回傳 *z* 的雙曲正弦值。
`tanh(z)`	回傳 *z* 的雙曲正切值。
分類函數	
`isfinite(z)`	Check if all components of *z* are finite
`isinf(z)`	Check if any component of *z* is infinite
`isnan(z)`	Check if any component of *z* is a NaN
`isclose(a, b, *, rel_tol, abs_tol)`	檢查 *a* 和 *b* 的值是否接近
常數	
`pi`	$\pi = 3.141592...$
`e`	$e = 2.718281...$
`tau`	$\tau = 2\pi = 6.283185...$
`inf`	Positive infinity
`infj`	Pure imaginary infinity
`nan`	”Not a number” (NaN)
`nanj`	Pure imaginary NaN

9.3.1 轉𠀀到極座標和從極座標做轉𠀀

Python 𠀀數 z 是用 直角坐標或 笛卡爾坐標儲存在𠀀部的。它完全是由其 實部 `z.real` 和 𠀀部 `z.imag` 所𠀀定。

極座標提供了另一種表示𠀀數的方法。在極座標中，𠀀數 *z* 由𠀀對值 (modulus) *r* 和相位角 (phase) *phi* 定義。𠀀對值 *r* 是從 *z* 到原點的距離，而相位角 *phi* 是從正 x 軸到連接原點到 *z* 的𠀀段的逆時針角度（以弧度𠀀單位）。

以下的函式可用於原始直角座標與極座標之間的相互轉𠀀。

`cmath.`**`phase`**`(z)`

以浮點數的形式回傳 *z* 的相位角（也稱𠀀 *z* 的 引數）。`phase(z)` 等價於 `math.atan2(z.imag, z.real)`。結果將位於 [-*π*, *π*] 的範圍𠀀，且此操作的分枝切割將位於負實軸上。結果的符號會與 `z.imag` 的符號相同，即使 `z.imag` 𠀀零：

```
>>> phase(-1+0j)
3.141592653589793
>>> phase(-1-0j)
-3.141592653589793
```

備𠀀

𠀀數 *z* 的𠀀對值可以使用𠀀建的*`abs()`* 函式計算。𠀀有單獨的*`cmath`* 模組函式適用於此操作。

`cmath.`**`polar`**`(z)`

回傳 *z* 在極座標中的表達方式。回傳一組數對 `(r, phi)`，*r* 是 *z* 的𠀀對值，*phi* 是 *z* 的相位角。`polar(z)` 相當於 `(abs(z), phase(z))`。

`cmath.`**`rect`**`(r, phi)`

透過極座標 *r* 和 *phi* 回傳𠀀數 *z*。相當於 `complex(r * math.cos(phi), r * math.sin(phi))`。

9.3.2 🄵函數和對數函數

cmath.**exp**(*z*)

回傳 *e* 的 *z* 次方，其中 *e* 是自然對數的底數。

cmath.**log**(*z*[, *base*])

回傳 *z* 給定 *base* 的對數。如果未指定 *base*，則傳回 *z* 的自然對數。存在一條分枝切割，從 0 沿負實數軸到 -∞。

cmath.**log10**(*z*)

回傳 *z* 以 10 🄵底的對數。它與*log()* 具有相同的分枝切割。

cmath.**sqrt**(*z*)

回傳 *z* 的平方根。它與*log()* 具有相同的分枝切割。

9.3.3 三角函數

cmath.**acos**(*z*)

回傳 *z* 的反余弦值。存在兩條分枝切割：一條是從 1 沿著實數軸向右延伸到 ∞。另一條從 -1 沿實數軸向左延伸到 -∞。

cmath.**asin**(*z*)

回傳 *z* 的反正弦值。它與*acos()* 具有相同的分枝切割。

cmath.**atan**(*z*)

回傳 *z* 的反正切值。有兩條分枝切割：一條是從 `1j` 沿著🄵軸延伸到 `∞j`。另一條從 `-1j` 沿著🄵軸延伸到 `-∞j`。

cmath.**cos**(*z*)

回傳 *z* 的余弦值。

cmath.**sin**(*z*)

回傳 *z* 的正弦值。

cmath.**tan**(*z*)

回傳 *z* 的正切值。

9.3.4 雙曲函數

cmath.**acosh**(*z*)

回傳 *z* 的反雙曲余弦值。存在一條分枝切割，從 1 沿實數軸向左延伸到 -∞。

cmath.**asinh**(*z*)

回傳 *z* 的反雙曲正弦值。存在兩條分枝切割：一條是從 `1j` 沿著🄵軸延伸到 `∞j`。另一條從 `-1j` 沿著🄵軸延伸到 `-∞j`。

cmath.**atanh**(*z*)

回傳 *z* 的反雙曲正切值。存在兩條分枝切割：一條是從 `1` 沿著實數軸延伸到 ∞。另一條從 `-1` 沿著實數軸延伸到 `-∞`。

cmath.**cosh**(*z*)

回傳 *z* 的反雙曲余弦值。

cmath.**sinh**(*z*)

回傳 *z* 的反雙曲正弦值。

cmath.**tanh**(*z*)

回傳 *z* 的反雙曲正切值。

9.3.5 分類函式

cmath.**isfinite**(*z*)

如果 *z* 的實部和🄵部都是有限的，則回傳 `True`，否則回傳 `False`。

在 3.2 版被加入.

cmath.**isinf**(*z*)

如果 *z* 的實部或🄵部是無窮大，則回傳 `True`，否則回傳 `False`。

cmath.**isnan**(*z*)

如果 *z* 的實部或🄵部🄵 NaN，則回傳 `True`，否則回傳 `False`。

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

如果 *a* 和 *b* 的值相互接近，則回傳 `True`，否則回傳 `False`。

兩數是否足🄵接近取🄵於給定的🄵對及相對容許偏差 (tolerance)。如果🄵有錯誤發生，結果將🄵: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

rel_tol 🄵相對容許偏差 —— *a* 與 *b* 兩數差的最大容許值，與 *a* 及 *b* 兩數的🄵對值中較大者相關。例如欲設置 5% 的容許偏差，則傳入 `rel_tol=0.05`。其預設值🄵 `1e-09`，該值可確保兩數於大約 9 個十進數位🄵相同。*rel_tol* 須不🄵負且小於 `1.0`。

abs_tol is the absolute tolerance; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol * abs(x)`, which is `False` for any `x` and rel_tol less than `1.0`. So add an appropriate positive abs_tol argument to the call.

定義於 IEEE 754 浮點標准中的特殊值 `NaN`、`inf` 和 `-inf` 會根據該標准處理。更明確地🄵，`NaN` 不會與包含自身在🄵的任何數字足🄵接近，而 `inf` 及 `-inf` 皆只與自身接近。

在 3.5 版被加入.

> **也參考**
>
> **PEP 485** —— 用於測試近似相等的函式

9.3.6 常數

cmath.**pi**

數學常數 π，作🄵一個浮點數。

cmath.**e**

數學常數 e，作🄵一個浮點數。

cmath.**tau**

數學常數 τ，作🄵一個浮點數。

在 3.6 版被加入.

cmath.**inf**

正無窮大的浮點數。相當於 `float('inf')`。

在 3.6 版被加入.

cmath.**infj**

實部🄵零和🄵部🄵正無窮的🄵數。相當於 `complex(0.0, float('inf'))`。

在 3.6 版被加入.

cmath.**nan**

浮點「非數字」(NaN) 值。相當於 `float('nan')`。

在 3.6 版被加入.

cmath.**nanj**

實部🄵零和🄵部🄵 NaN 的🄵數。相當於 `complex(0.0, float('nan'))`。

在 3.6 版被加入.

請注意，函式的選擇與模組*math*的類似，但🄵不完全相同。擁有兩個模組的原因是有些用🄵對🄵數不感興趣，甚至根本就不知道它們是什🄵。他們寧願讓 `math.sqrt(-1)` 引發🄵常，也不願它回傳🄵數。另請注意，*cmath* 中所定義的函式始終都會回傳🄵數，即使答案可以表示🄵實數（在這種情🄵下，🄵數的🄵部🄵零）。

關於分枝切割的🄵釋：它們是沿著給定的不連續函式的曲🄵。它們是許多🄵變函數的必要特徵。假設你需要使用🄵變函數進行計算，你將會了解分枝切割的概念。請參🄵幾乎所有關於🄵變函數的（不是太初級的）書籍以獲得🄵發。對於如何正確地基於數值目的選擇分枝切割的相關訊息，以下🄵容應該是一個很好的參考：

也參考

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165--211.

9.4 `decimal` --- 十進位固定點和浮點運算

原始碼： Lib/decimal.py

The *decimal* module provides support for fast correctly rounded decimal floating-point arithmetic. It offers several advantages over the *float* datatype:

- Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle -- computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." -- excerpt from the decimal arithmetic specification.
- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.
- The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
- The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.
- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

- The decimal module was designed to support "without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic." -- excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include *`ROUND_CEILING`*, *`ROUND_DOWN`*, *`ROUND_FLOOR`*, *`ROUND_HALF_DOWN`*, *`ROUND_HALF_EVEN`*, *`ROUND_HALF_UP`*, *`ROUND_UP`*, and *`ROUND_05UP`*.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: *`Clamped`*, *`InvalidOperation`*, *`DivisionByZero`*, *`Inexact`*, *`Rounded`*, *`Subnormal`*, *`Overflow`*, *`Underflow`* and *`FloatOperation`*.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

也參考

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Quick-start tutorial

The usual start to using decimals is importing the module, viewing the current context with *`getcontext()`* and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for "Not a number", positive and negative `Infinity`, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

If the *FloatOperation* signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

在 3.3 版被加入.

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises *InvalidOperation*:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

在 3.3 版的變更.

Decimals interact well with much of the rest of Python. Here is a small decimal floating-point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
```

(繼續下一頁)

(繼續上一頁)

```
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

The *quantize()* method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the *getcontext()* function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the Context() constructor. To make an alternate active, use the *setcontext()* function.

In accordance with the standard, the *decimal* module provides two ready to use standard contexts, *BasicContext* and *ExtendedContext*. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the *clear_flags()* method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the *traps* attribute of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to *Decimal* with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

9.4.2 Decimal objects

class decimal.**Decimal**(*value='0'*, *context=None*)

Construct a new *Decimal* object based from *value*.

value can be an integer, string, tuple, *float*, or another *Decimal* object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`. Case is not significant, so, for example, `inf`, `Inf`, `INFINITY`, and `iNfINity` are all acceptable spellings for positive infinity.

If *value* is a *tuple*, it should have three components, a sign (`0` for positive or `1` for negative), a *tuple* of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If *value* is a *float*, the binary floating-point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.100000000000000088817841970012523233890533447265625')`.

The *context* precision does not affect how many digits are stored. That is determined exclusively by the number of digits in *value*. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps *InvalidOperation*, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, *Decimal* objects are immutable.

在 3.2 版的變更: The argument to the constructor is now permitted to be a *float* instance.

在 3.3 版的變更: *float* arguments raise an exception if the *FloatOperation* trap is set. By default the trap is off.

在 3.6 版的變更: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating-point objects share many properties with the other built-in numeric types such as *float* and *int*. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as *float* or *int*).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator `%` is applied to Decimal objects, the sign of the result is the sign of the *dividend* rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The `%` and `//` operators implement the `remainder` and `divide-integer` operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of *fractions.Fraction* in arithmetic operations: an attempt to add a *Decimal* to a *float*, for example, will raise a *TypeError*. However, it is possible to use Python's comparison operators to compare a *Decimal* instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types.

在 3.2 版的變更: Mixed-type comparisons between *Decimal* instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating-point objects also have a number of specialized methods:

adjusted()

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()

Return a pair `(n, d)` of integers that represent the given *Decimal* instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

The conversion is exact. Raise OverflowError on infinities and ValueError on NaNs.

在 3.6 版被加入.

as_tuple()

Return a *named tuple* representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a *Decimal* instance is always canonical, so this operation returns its argument unchanged.

compare(*other*, *context=None*)

Compare the values of two Decimal instances. *compare()* returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

This operation is identical to the *compare()* method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare_total(*other*, *context=None*)

Compare two operands using their abstract representation rather than their numerical value. Similar to the *compare()* method, but the result gives a total ordering on *Decimal* instances. Two *Decimal* instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

compare_total_mag(*other*, *context=None*)

Compare two operands using their abstract representation rather than their value as in *compare_total()*, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

conjugate()

Just returns self, this method is only to comply with the Decimal Specification.

copy_abs()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_negate()

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_sign(*other*, *context=None*)

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

exp(*context=None*)

Return the value of the (natural) exponential function `e**x` at the given number. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*, */*)

Alternative constructor that only accepts instances of *float* or *int*.

Note `Decimal.from_float(0.1)` is not the same as `Decimal('0.1')`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is `0x1.999999999999ap-4`. That equivalent value in decimal is `0.1000000000000000055511151231257827021181583404541015625`.

> **備🄵**
>
> From Python 3.2 onwards, a *Decimal* instance can also be constructed directly from a *float*.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

在 3.1 版被加入.

classmethod from_number(*number*, */*)

Alternative constructor that only accepts instances of *float*, *int* or *Decimal*, but not strings or tuples.

```
>>> Decimal.from_number(314)
Decimal('314')
>>> Decimal.from_number(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_number(Decimal('3.14'))
Decimal('3.14')
```

在 3.14 版被加入.

fma(*other*, *third*, *context=None*)

Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

Return *True* if the argument is canonical and *False* otherwise. Currently, a *Decimal* instance is always canonical, so this operation always returns *True*.

is_finite()

Return *True* if the argument is a finite number, and *False* if the argument is an infinity or a NaN.

is_infinite()

Return *True* if the argument is either positive or negative infinity and *False* otherwise.

is_nan()

Return *True* if the argument is a (quiet or signaling) NaN and *False* otherwise.

is_normal(*context=None*)

Return *True* if the argument is a *normal* finite number. Return *False* if the argument is zero, subnormal, infinite or a NaN.

is_qnan()

Return *True* if the argument is a quiet NaN, and *False* otherwise.

is_signed()

Return *True* if the argument has a negative sign and *False* otherwise. Note that zeros and NaNs can both carry signs.

is_snan()

Return *True* if the argument is a signaling NaN and *False* otherwise.

is_subnormal(*context=None*)

Return *True* if the argument is subnormal, and *False* otherwise.

is_zero()

Return *True* if the argument is a (positive or negative) zero and *False* otherwise.

ln(*context=None*)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

log10(*context=None*)

Return the base ten logarithm of the operand. The result is correctly rounded using the *ROUND_HALF_EVEN* rounding mode.

logb(*context=None*)

For a nonzero number, return the adjusted exponent of its operand as a *Decimal* instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the *DivisionByZero* flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

logical_and(*other*, *context=None*)

logical_and() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `and` of the two operands.

logical_invert(*context=None*)

logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or(*other*, *context=None*)

logical_or() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `or` of the two operands.

logical_xor(*other*, *context=None*)

logical_xor() is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise exclusive or of the two operands.

max(*other*, *context=None*)

Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(*other*, *context=None*)

Similar to the *max()* method, but the comparison is done using the absolute values of the operands.

min(*other*, *context=None*)

Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(*other*, *context=None*)

Similar to the *min()* method, but the comparison is done using the absolute values of the operands.

next_minus(*context=None*)

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

next_plus(*context=None*)

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

next_toward(*other*, *context=None*)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize(*context=None*)

Used for producing canonical values of an equivalence class within either the current context or the specified context.

This has the same semantics as the unary plus operation, except that if the final result is finite it is reduced to its simplest form, with all trailing zeros removed and its sign preserved. That is, while the coefficient is non-zero and a multiple of ten the coefficient is divided by ten and the exponent is incremented by 1. Otherwise (the coefficient is zero) the exponent is set to 0. In all cases the sign is unchanged.

For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

Note that rounding is applied *before* reducing to simplest form.

In the latest versions of the specification, this operation is also known as `reduce`.

number_class(*context=None*)

Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.
- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

quantize(*exp*, *rounding=None*, *context=None*)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an *InvalidOperation* is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread's context is used.

An error is returned whenever the resulting exponent is greater than *Emax* or less than *Etiny()*.

radix()

Return `Decimal(10)`, the radix (base) in which the *Decimal* class does all its arithmetic. Included for compatibility with the specification.

remainder_near(*other*, *context=None*)

Return the remainder from dividing *self* by *other*. This differs from `self % other` in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

same_quantum(*other*, *context=None*)

Test whether self and other have the same exponent or whether both are `NaN`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

scaleb(*other*, *context=None*)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

shift(*other*, *context=None*)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt(*context=None*)

Return the square root of the argument to full precision.

to_eng_string(*context=None*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)

Identical to the *to_integral_value()* method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact(*rounding=None*, *context=None*)

Round to the nearest integer, signaling *Inexact* or *Rounded* as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value(*rounding=None*, *context=None*)

Round to the nearest integer without signaling *Inexact* or *Rounded*. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied *context* or the current context.

Decimal numbers can be rounded using the *round()* function:

round(number)

round(number, ndigits)

If *ndigits* is not given or `None`, returns the nearest *int* to *number*, rounding ties to even, and ignoring the rounding mode of the *Decimal* context. Raises *OverflowError* if *number* is an infinity or *ValueError* if it is a (quiet or signaling) NaN.

If *ndigits* is an *int*, the context's rounding mode is respected and a *Decimal* representing *number* rounded to the nearest multiple of `Decimal('1E-ndigits')` is returned; in this case, `round(number, ndigits)` is equivalent to `self.quantize(Decimal('1E-ndigits'))`. Returns `Decimal('NaN')` if *number* is a quiet NaN. Raises *InvalidOperation* if *number* is an infinity, a signaling NaN, or if the length of the coefficient after the quantize operation would be greater than the current context's precision. In other words, for the non-corner cases:

- if *ndigits* is positive, return *number* rounded to *ndigits* decimal places;
- if *ndigits* is zero, return *number* rounded to the nearest integer;
- if *ndigits* is negative, return *number* rounded to the nearest multiple of `10**abs(ndigits)`.

For example:

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))      # context rounding ignored
4
>>> round(Decimal('3.5'))       # round-ties-to-even
4
>>> round(Decimal('3.75'), 0)   # uses the context rounding
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

Logical operands

The *logical_and()*, *logical_invert()*, *logical_or()*, and *logical_xor()* methods expect their arguments to be *logical operands*. A *logical operand* is a *Decimal* instance whose exponent and sign are both zero, and whose digits are all either `0` or `1`.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the *getcontext()* and *setcontext()* functions:

decimal.**getcontext**()

Return the current context for the active thread.

decimal.**setcontext**(*c*, /)

Set the current context for the active thread to *c*.

You can also use the `with` statement and the *localcontext()* function to temporarily change the active context.

decimal.**localcontext**(*ctx=None*, ***kwargs*)

Return a context manager that will set the current context for the active thread to a copy of *ctx* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. The *kwargs* argument is used to set the attributes of the new context.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

Using keyword arguments, the code would be the following:

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
    s = calculate_something()
s = +s
```

Raises *TypeError* if *kwargs* supplies an attribute that *Context* doesn't support. Raises either *TypeError* or *ValueError* if *kwargs* supplies an invalid value for an attribute.

在 3.11 版的變更: *localcontext()* now supports setting context attributes through the use of keyword arguments.

decimal.**IEEEContext**(*bits*)

Return a context object initialized to the proper values for one of the IEEE interchange formats. The argument must be a multiple of 32 and less than *IEEE_CONTEXT_MAX_BITS*.

在 3.14 版被加入.

New contexts can also be created using the *Context* constructor described below. In addition, the module provides three pre-made contexts:

decimal.**BasicContext**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_UP*. All flags are cleared. All traps are enabled (treated as exceptions) except *Inexact*, *Rounded*, and *Subnormal*.

Because many of the traps are enabled, this context is useful for debugging.

decimal.**ExtendedContext**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to *ROUND_HALF_EVEN*. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

`decimal.`**`DefaultContext`**

This context is used by the *`Context`* constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the *`Context`* constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are *`Context.prec`*=`28`, *`Context.rounding`*=*`ROUND_HALF_EVEN`*, and enabled traps for *`Overflow`*, *`InvalidOperation`*, and *`DivisionByZero`*.

In addition to the three supplied contexts, new contexts can be created with the *`Context`* constructor.

class `decimal.`**`Context`**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Creates a new context. If a field is not specified or is *`None`*, the default values are copied from the *`DefaultContext`*. If the *flags* field is not specified or is *`None`*, all flags are cleared.

`prec`

An integer in the range [`1`, *`MAX_PREC`*] that sets the precision for arithmetic operations in the context.

`rounding`

One of the constants listed in the section *Rounding Modes*.

`traps`

`flags`

Lists of any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

`Emin`

`Emax`

Integers specifying the outer limits allowable for exponents. *Emin* must be in the range [*`MIN_EMIN`*, `0`], *Emax* in the range [`0`, *`MAX_EMAX`*].

`capitals`

Either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

`clamp`

Either `0` (the default) or `1`. If set to `1`, the exponent `e` of a *`Decimal`* instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If *clamp* is `0` then a weaker condition holds: the adjusted exponent of the *`Decimal`* instance is at most *`Emax`*. When *clamp* is `1`, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of `1` allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The *`Context`* class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the *`Decimal`* methods described above (with the exception of the *`adjusted()`* and *`as_tuple()`* methods) there is a corresponding *`Context`* method. For example, for a *`Context`* instance `C` and *`Decimal`* instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`.

Each *Context* method accepts a Python integer (an instance of *int*) anywhere that a Decimal instance is accepted.

clear_flags()

Resets all of the flags to `0`.

clear_traps()

Resets all of the traps to `0`.

在 3.3 版被加入.

copy()

Return a duplicate of the context.

copy_decimal(*num*, /)

Return a copy of the Decimal instance num.

create_decimal(*num='0'*, /)

Creates a new Decimal instance from *num* but using *self* as context. Unlike the *Decimal* constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

create_decimal_from_float(*f*, /)

Creates a new Decimal instance from a float *f* but rounding using *self* as the context. Unlike the *Decimal.from_float()* class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

在 3.1 版被加入.

Etiny()

Returns a value equal to `Emin - prec + 1` which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to *Etiny*.

Etop()

Returns a value equal to `Emax - prec + 1`.

The usual approach to working with decimals is to create *Decimal* instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the *Decimal* class and are only briefly recounted here.

abs(*x*, /)
Returns the absolute value of *x*.

add(*x*, *y*, /)
Return the sum of *x* and *y*.

canonical(*x*, /)
Returns the same Decimal object *x*.

compare(*x*, *y*, /)
Compares *x* and *y* numerically.

compare_signal(*x*, *y*, /)
Compares the values of the two operands numerically.

compare_total(*x*, *y*, /)
Compares two operands using their abstract representation.

compare_total_mag(*x*, *y*, /)
Compares two operands using their abstract representation, ignoring sign.

copy_abs(*x*, /)
Returns a copy of *x* with the sign set to 0.

copy_negate(*x*, /)
Returns a copy of *x* with the sign inverted.

copy_sign(*x*, *y*, /)
Copies the sign from *y* to *x*.

divide(*x*, *y*, /)
Return *x* divided by *y*.

divide_int(*x*, *y*, /)
Return *x* divided by *y*, truncated to an integer.

divmod(*x*, *y*, /)
Divides two numbers and returns the integer part of the result.

exp(*x*, /)
Returns `e ** x`.

fma(*x*, *y*, *z*, /)
Returns *x* multiplied by *y*, plus *z*.

is_canonical(*x*, /)
Returns `True` if *x* is canonical; otherwise returns `False`.

is_finite(*x*, /)
Returns `True` if *x* is finite; otherwise returns `False`.

is_infinite(*x*, /)
Returns `True` if *x* is infinite; otherwise returns `False`.

is_nan(*x*, /)
Returns `True` if *x* is a qNaN or sNaN; otherwise returns `False`.

is_normal(*x*, /)
Returns `True` if *x* is a normal number; otherwise returns `False`.

is_qnan(*x*, /)
Returns `True` if *x* is a quiet NaN; otherwise returns `False`.

is_signed(*x*, /)

Returns `True` if *x* is negative; otherwise returns `False`.

is_snan(*x*, /)

Returns `True` if *x* is a signaling NaN; otherwise returns `False`.

is_subnormal(*x*, /)

Returns `True` if *x* is subnormal; otherwise returns `False`.

is_zero(*x*, /)

Returns `True` if *x* is a zero; otherwise returns `False`.

ln(*x*, /)

Returns the natural (base e) logarithm of *x*.

log10(*x*, /)

Returns the base 10 logarithm of *x*.

logb(*x*, /)

Returns the exponent of the magnitude of the operand's MSD.

logical_and(*x*, *y*, /)

Applies the logical operation *and* between each operand's digits.

logical_invert(*x*, /)

Invert all the digits in *x*.

logical_or(*x*, *y*, /)

Applies the logical operation *or* between each operand's digits.

logical_xor(*x*, *y*, /)

Applies the logical operation *xor* between each operand's digits.

max(*x*, *y*, /)

Compares two values numerically and returns the maximum.

max_mag(*x*, *y*, /)

Compares the values numerically with their sign ignored.

min(*x*, *y*, /)

Compares two values numerically and returns the minimum.

min_mag(*x*, *y*, /)

Compares the values numerically with their sign ignored.

minus(*x*, /)

Minus corresponds to the unary prefix minus operator in Python.

multiply(*x*, *y*, /)

Return the product of *x* and *y*.

next_minus(*x*, /)

Returns the largest representable number smaller than *x*.

next_plus(*x*, /)

Returns the smallest representable number larger than *x*.

next_toward(*x*, *y*, /)

Returns the number closest to *x*, in direction towards *y*.

normalize(*x*, /)

Reduces *x* to its simplest form.

number_class(*x*, */*)

Returns an indication of the class of *x*.

plus(*x*, */*)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

power(*x*, *y*, *modulo=None*)

Return `x` to the power of `y`, reduced modulo `modulo` if given.

With two arguments, compute `x**y`. If `x` is negative then `y` must be integral. The result will be inexact unless `y` is integral and the result is finite and can be expressed exactly in 'precision' digits. The rounding mode of the context is used. Results are always correctly rounded in the Python version.

`Decimal(0) ** Decimal(0)` results in `InvalidOperation`, and if `InvalidOperation` is not trapped, then results in `Decimal('NaN')`.

在 3.3 版的變更: The C module computes *power()* in terms of the correctly rounded *exp()* and *ln()* functions. The result is well-defined but only "almost always correctly rounded".

With three arguments, compute `(x**y) % modulo`. For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- `y` must be nonnegative
- at least one of `x` or `y` must be nonzero
- `modulo` must be nonzero and have at most 'precision' digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y` and `modulo`. The result is always exact.

quantize(*x*, *y*, */*)

Returns a value equal to *x* (rounded), having the exponent of *y*.

radix()

Just returns 10, as this is Decimal, :)

remainder(*x*, *y*, */*)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near(*x*, *y*, */*)

Returns `x - y * n`, where *n* is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of *x*).

rotate(*x*, *y*, */*)

Returns a rotated copy of *x*, *y* times.

same_quantum(*x*, *y*, */*)

Returns `True` if the two operands have the same exponent.

scaleb(*x*, *y*, */*)

Returns the first operand after adding the second value its exp.

shift(*x*, *y*, */*)

Returns a shifted copy of *x*, *y* times.

sqrt(*x*, */*)

Square root of a non-negative number to context precision.

subtract(*x*, *y*, */*)

Return the difference between *x* and *y*.

to_eng_string(*x*, */*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

to_integral_exact(*x*, */*)

Rounds to an integer.

to_sci_string(*x*, */*)

Converts a number to a string using scientific notation.

9.4.4 常數

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32 位元	64 位元
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997
decimal.**IEEE_CONTEXT_MAX_BITS**	256	512

decimal.**HAVE_THREADS**

The value is `True`. Deprecated, because Python now always has threads.

在 3.9 版之後被🄵用.

decimal.**HAVE_CONTEXTVAR**

The default value is `True`. If Python is `configured using the --without-decimal-contextvar option`, the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

在 3.8.3 版被加入.

9.4.5 Rounding modes

decimal.**ROUND_CEILING**

Round towards `Infinity`.

decimal.**ROUND_DOWN**

Round towards zero.

decimal.**ROUND_FLOOR**

Round towards `-Infinity`.

decimal.**ROUND_HALF_DOWN**

Round to nearest with ties going towards zero.

decimal.**ROUND_HALF_EVEN**

Round to nearest with ties going to nearest even integer.

decimal.**ROUND_HALF_UP**

Round to nearest with ties going away from zero.

decimal.**ROUND_UP**

Round away from zero.

decimal.**ROUND_05UP**

Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the *DivisionByZero* trap is set, then a *DivisionByZero* exception is raised upon encountering the condition.

class decimal.**Clamped**

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's *Emin* and *Emax* limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.**DecimalException**

Base class for other signals and a subclass of *ArithmeticError*.

class decimal.**DivisionByZero**

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.**Inexact**

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.**InvalidOperation**

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
```

(繼續下一頁)

(繼續上一頁)

```
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**

Numerical overflow.

Indicates the exponent is larger than *Context.Emax* after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.**Rounded**

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.**Subnormal**

Exponent was lower than *Emin* prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.**Underflow**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. *Inexact* and *Subnormal* are also signaled.

class decimal.**FloatOperation**

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating-point notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating-point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The *decimal* module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊值

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the *DivisionByZero* signal is not trapped. Likewise, when the *Overflow* signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the *InvalidOperation* signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means "not a number". This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs --- it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns *False* (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns *True*. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the *InvalidOperation* signal if either operand is a `NaN`, and return *False* if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from

the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the *compare()* and *compare_signal()* methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating-point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Working with threads

The *getcontext()* function accesses a different *Context* object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the *setcontext()* function automatically assigns its target to the current thread.

If *setcontext()* has not been called before *getcontext()*, then *getcontext()* will automatically create a new context for use in the current thread. New context objects have default values set from the *decimal.DefaultContext* object.

The *sys.flags.thread_inherit_context* flag affects the context for new threads. If the flag is false, new threads will start with an empty context. In this case, *getcontext()* will create a new context object when called and use the default values from *DefaultContext*. If the flag is true, new threads will start with a copy of context from the caller of *threading.Thread.start()*.

To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won't be a race condition between threads calling *getcontext()*. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the *Decimal* class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
```

(繼續下一頁)

(繼續上一頁)

```
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.
```

(繼續下一頁)

(繼續上一頁)

```
    >>> print(exp(Decimal(1)))
    2.71828182845904523536028747 1
    >>> print(exp(Decimal(2)))
    7.38905609893065022723042746 1
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
```

(繼續下一頁)

(繼續上一頁)

```
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The *quantize()* method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a *quantize()* step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
```

(繼續下一頁)

(繼續上一頁)

```
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the *quantize()* step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                           # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `.02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The *normalize()* method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. When does rounding occur in a computation?

A. It occurs *after* the computation. The philosophy of the decimal specification is that numbers are considered exact and are created independent of the current context. They can even have greater precision than current context. Computations process with those exact inputs and then rounding (or other context operations) is applied to the *result* of the computation:

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')    # More than 5 digits
>>> pi                              # All digits are retained
Decimal('3.1415926535')
>>> pi + 0                          # Rounded after an addition
Decimal('3.1416')
>>> pi - Decimal('0.00005')         # Subtract unrounded numbers, then round
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').    # Intermediate values are rounded
Decimal('3.1416')
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a *Decimal*?

A. Yes, any binary floating-point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the *Context.create_decimal()* method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. Is the CPython implementation fast for large numbers?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec library for arbitrary precision correctly rounded decimal floating-point arithmetic[1]. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. *Emin* and *Emax* should always be set to the maximum values, *clamp* should always be 0 (the default). Setting *prec* requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for *prec* as well[2]:

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal('904625697166532776746648320380374280103671755200316906558262375061821325312')
```

For inexact results, *MAX_PREC* is far too large on 64-bit platforms and the available memory will be insufficient:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust *prec* to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each:

[1] 在 3.3 版被加入.

[2] 在 3.9 版的變更: This approach now works for all exact results except for non-integer powers.

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

In general (and especially on systems without overallocation), it is recommended to estimate even tighter bounds and set the *Inexact* trap if all calculations are expected to be exact.

9.5 fractions --- 有理數

原始碼： Lib/fractions.py

The *fractions* module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

class fractions.**Fraction**(*numerator=0*, *denominator=1*)

class fractions.**Fraction**(*number*)

class fractions.**Fraction**(*string*)

The first version requires that *numerator* and *denominator* are instances of *numbers.Rational* and returns a new *Fraction* instance with value `numerator/denominator`. If *denominator* is `0`, it raises a *ZeroDivisionError*.

The second version requires that *number* is an instance of *numbers.Rational* or has the `as_integer_ratio()` method (this includes *float* and *decimal.Decimal*). It returns a *Fraction* instance with exactly the same value. Assumed, that the `as_integer_ratio()` method returns a pair of coprime integers and last one is positive. Note that due to the usual issues with binary point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the *limit_denominator()* method below.)

The last version of the constructor expects a string. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either '+' or '-' and `numerator` and `denominator` (if present) are strings of decimal digits (underscores may be used to delimit digits as with integral literals in code). In addition, any string that represents a finite value and is accepted by the *float* constructor is also accepted by the *Fraction* constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
```

(繼續下一頁)

(繼續上一頁)

```
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are *hashable*, and should be treated as immutable. In addition, *Fraction* has the following properties and methods:

在 3.2 版的變更: *Fraction* 建構函式現在可接受*float* 和*decimal.Decimal* 實例。

在 3.9 版的變更: The *math.gcd()* function is now used to normalize the *numerator* and *denominator*. *math.gcd()* always returns an *int* type. Previously, the GCD type depended on *numerator* and *denominator*.

在 3.11 版的變更: Underscores are now permitted when creating a *Fraction* instance from a string, following **PEP 515** rules.

在 3.11 版的變更: *Fraction* implements `__int__` now to satisfy `typing.SupportsInt` instance checks.

在 3.12 版的變更: Space is allowed around the slash for string inputs: `Fraction('2 / 3')`.

在 3.12 版的變更: *Fraction* instances now support float-style formatting, with presentation types `"e"`, `"E"`, `"f"`, `"F"`, `"g"`, `"G"` and `"%"`.

在 3.13 版的變更: Formatting of *Fraction* instances without a presentation type now supports fill, alignment, sign handling, minimum width and grouping.

在 3.14 版的變更: *Fraction* 建構函式現在可接受任何具有 `as_integer_ratio()` 方法的物件。

numerator

Numerator of the Fraction in lowest term.

denominator

Denominator of the Fraction in lowest term.

as_integer_ratio()

Return a tuple of two integers, whose ratio is equal to the original Fraction. The ratio is in lowest terms and has a positive denominator.

在 3.8 版被加入.

is_integer()

Return `True` if the Fraction is an integer.

在 3.12 版被加入.

classmethod from_float(*f*)

Alternative constructor which only accepts instances of *float* or *numbers.Integral*. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

> **備🄵**
>
> From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *float*.

classmethod from_decimal(*dec*)

Alternative constructor which only accepts instances of *decimal.Decimal* or *numbers.Integral*.

> **備🄵**
>
> From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *decimal.Decimal* instance.

classmethod from_number(*number*)

Alternative constructor which only accepts instances of *numbers.Integral*, *numbers.Rational*, *float* or *decimal.Decimal*, and objects with the `as_integer_ratio()` method, but not strings.

在 3.14 版被加入.

limit_denominator(*max_denominator=1000000*)

Finds and returns the closest *Fraction* to `self` that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

Returns the greatest *int* `<= self`. This method can also be accessed through the *math.floor()* function:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Returns the least *int* `>= self`. This method can also be accessed through the *math.ceil()* function.

__round__()

__round__(*ndigits*)

The first version returns the nearest *int* to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the *round()* function.

__format__(*format_spec*, */*)

Provides support for formatting of `Fraction` instances via the `str.format()` method, the `format()` built-in function, or Formatted string literals.

If the `format_spec` format specification string does not end with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the general rules for fill, alignment, sign handling, minimum width, and grouping as described in the *format specification mini-language*. The "alternate form" flag `'#'` is supported: if present, it forces the output string to always include an explicit denominator, even when the value being formatted is an exact integer. The zero-fill flag `'0'` is not supported.

If the `format_spec` format specification string ends with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the rules outlined for the `float` type in the 格式規格 *(Format Specification)* 迷你語言 section.

Here are some examples:

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price, old_price) - 1)
'34.67% price increase'
```

也參考

***numbers* 模組**
The abstract base classes making up the numeric tower.

9.6 random --- 🄵生🄵隨機數

原始碼： Lib/random.py

本章中所提及的 module（模組）用來實現各種分🄵的🄵擬隨機數🄵生器。

對於整數，可以從範圍中進行均🄵選擇。對於序列，有一個隨機元素的均🄵選擇，一個用來原地 (in-place) 🄵生隨機排列清單的函式，以及一個用來隨機取樣不重置的函式。

在實數🄵上，有一些函式用於處理均🄵分🄵、常態分🄵（高斯分🄵）、對數常態分🄵、負指數分🄵、gamma 分🄵和 Beta 分🄵。對於🄵生角度分🄵，可以使用馮．米塞斯分🄵 (von Mises distribution)。

幾乎所有 module 函式都相依於基本函式`random()`，此函式在半開放範圍 `0.0 <= X < 1.0` 🄵均🄵地🄵生一個隨機 float（浮點數）。Python 使用 Mersenne Twister（梅森旋轉演算法）作🄵核心的🄵生器，它🄵生 53 位元精度 float，其🄵期🄵 2**19937-1，透過 C 語言進行底層的實作既快速又支援執行緒安全 (threadsafe)。Mersenne Twister 是現存最廣泛被驗證的隨機數🄵生器之一，但是基於完全確定性，它🄵不適合所有目的，🄵且完全不適合加密目的。

該 module 提供的函式實際上是`random.Random` class（類🄵）中一個隱藏實例的綁定方法 (bound method)。你可以實例化自己的`Random` 實例，以得到不共享狀態的🄵生器。

如果你想使用你自己設計的基本🄵生器，*Random* 也可以進行子類🄵化 (subclass)。有關詳細資訊，請參考該類🄵的文件。

random module 也提供了 *SystemRandom* class，使用系統函式 *os.urandom()* 從作業系統提供的來源🄵生隨機數。

警告

本章所提及的🄵擬隨機數🄵生器不應該使用於安全目的。有關安全性或加密用途，請參考 *secrets* module。

也參考

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3--30 1998.

進位互補乘法 (Complementary-Multiply-with-Carry) 用法，可作🄵隨機數🄵生器的一個可相容替代方案，具有較長的🄵期和相對簡單的更新操作。

備🄵

全域隨機數🄵生器和 *Random* 的實例是執行緒安全的。然而，在自由執行緒構建中，對全域🄵生器或同一 *Random* 實例的🄵行呼叫可能會遇到🄵用和性能不佳。請考慮改🄵每個執行緒使用單獨的 *Random* 實例。

9.6.1 簿記函式 (bookkeeping functions)

`random.`**`seed`**`(`*a=None*, *version=2*`)`

初始化隨機數🄵生器。

如果 *a* 被省略或🄵 `None`，則使用目前系統時間。如果隨機來源由作業系統提供，則使用它們而不是系統時間（有關可用性的詳細資訊，請參考 *os.urandom()* 函式）。

如果 *a* 🄵 int（整數），則直接使用它。

如使用版本 2（預設值），*str*、*bytes* 或 *bytearray* 物件將轉🄵🄵 *int*，🄵使用其所有位元。

若使用版本 1（🄵復現於舊版本 Python 中🄵生隨機序列而提供），*str* 和 *bytes* 的演算法會🄵生範圍更窄的種子 (seed)。

在 3.2 版的變更: 移至版本 2 方案，該方案使用字串種子中的所有位元。

在 3.11 版的變更: *seed* 必須是以下型🄵之一：`None`、*int*、*float*、*str*、*bytes*、*bytearray*。

`random.`**`getstate`**`()`

回傳一個物件，捕獲🄵生器的目前🄵部狀態。此物件可以傳遞給 *setstate()* 以恢復狀態。

`random.`**`setstate`**`(`*state*`)`

state 應該要從之前對 *getstate()* 的呼叫中獲得，🄵且以 *setstate()* 將🄵生器的🄵部狀態恢復到呼叫 *getstate()* 時的狀態。

9.6.2 回傳位元組的函式

`random.`**`randbytes`**`(`*n*`)`

🄵生 *n* 個隨機位元組。

此方法不應使用於🄵生安全性權杖 (Token)。請改用 *secrets.token_bytes()*。

在 3.9 版被加入.

9.6.3 回傳整數的函式

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*[, *step*])

傳回從 `range(start, stop, step)` 中隨機選擇的元素。

這大致相當於 `choice(range(start, stop, step))`，但支援任意大的範圍，🄵針對常見情🄵進行了最佳化。

位置引數模式與*range()* 函式相符。

不應使用關鍵字引數，因🄵它們可能會以意想不到的方式被直譯。例如 `randrange(start=100)` 會被直譯🄵 `randrange(0, 100, 1)`。

在 3.2 版的變更: *randrange()* 在🄵生均🄵分🄵的值方面更🄵🄵雜。以前，它使用像 `int(random()*n)` 這樣的樣式，這可能會🄵生稍微不均🄵的分🄵。

在 3.12 版的變更: 已經不再支援非整數類型到等效整數的自動轉🄵。像是 `randrange(10.0)` 和 `randrange(Fraction(10, 1))` 的呼叫將會引發*TypeError*。

random.**randint**(*a*, *b*)

回傳一個隨機整數 *N*，使得 `a <= N <= b`。🄵 `randrange(a, b+1)` 的🄵名。

random.**getrandbits**(*k*)

回傳一個具有 *k* 個隨機位元的非負 Python 整數。此方法會隨 Mersenne Twister 🄵生器一起提供，一些其他的🄵生器也可能將其作🄵 API 的可選部分。如果可用，*getrandbits()* 使*randrange()* 能🄵處理任意大的範圍。

在 3.9 版的變更: 此方法現在接受 *k* 🄵零。

9.6.4 回傳序列的函式

random.**choice**(*seq*)

從非空序列 *seq* 回傳一個隨機元素。如果 *seq* 🄵空，則引發*IndexError*。

random.**choices**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)

回傳從 *population* 中重置取樣出的一個大小🄵 *k* 的元素 list。如果 *population* 🄵空，則引發*IndexError*。

如果指定了 *weights* 序列，則根據相對權重進行選擇。另外，如果給定 *cum_weights* 序列，則根據累積權重進行選擇（可能使用*itertools.accumulate()* 計算）。例如，相對權重 `[10, 5, 30, 5]` 等同於累積權重 `[10, 15, 45, 50]`。在🄵部，相對權重在進行選擇之前會轉🄵🄵累積權重，因此提供累積權重可以節省工作。

如果既未指定 *weights* 也未指定 *cum_weights*，則以相等的機率進行選擇。如果提供了加權序列，則該序列的長度必須與 *population* 序列的長度相同。它是一個*TypeError* 來指定 *weights* 和 *cum_weights*。

weights 或 *cum_weights* 可以使用任何與*random()* 所回傳的*float* 值（包括整數、float 和分數，但不包括小數）交互操作 (interoperates) 的數值類型。權重假定🄵非負數和有限的。如果所有權重均🄵零，則引發*ValueError*。

對於給定的種子，具有相等權重的*choices()* 函式通常🄵生與重🄵呼叫*choice()* 不同的序列。*choices()* 使用的演算法使用浮點運算來實現🄵部一致性和速度。*choice()* 使用的演算法預設🄵整數運算和重🄵選擇，以避免舍入誤差🄵生的小偏差。

在 3.6 版被加入.

在 3.9 版的變更: 如果所有權重均🄵零，則引發*ValueError*。

random.**shuffle**(*x*)

將序列 *x* 原地 (in place) 隨機打亂位置。

要打亂一個不可變的序列🄵回傳一個新的被打亂的 list（串列），請使用 `sample(x, k=len(x))`。

請注意，即使對於較小的 `len(x)`，*x* 的置🄴總數也會快速成長到大於大多數隨機數🄵生器的🄴期。這意味著長序列的大多數置🄴永遠無法🄴生。例如，長度🄵 2080 的序列是 Mersenne Twister 隨機數🄵生器🄴期🄴可以容納的最大序列。

在 3.11 版的變更: 移除可選參數 *random*。

`random.`**`sample`**(*population*, *k*, *, *counts=None*)

回傳從母體序列中選擇出的一個包含獨特元素、長度🄴 *k* 的 list。用於不重置的隨機取樣。

回傳包含母體元素的新清單，同時保持原始母體不變。🄴生的清單按選擇順序排列，因此所有子切片也會是有效的隨機樣本。這允許抽🄴獲🄴者（樣本）分🄴大🄴和第二名獲🄴者（子切片）。

母體成員不必是*hashable* 或唯一的。如果母體包含重🄴項，則每次出現都是樣本中可能出現的一個選擇。

可以一次指定一個重🄴元素，也可以使用可選的僅關鍵字 *counts* 參數指定重🄴元素。例如 `sample(['red', 'blue'], counts=[4, 2], k=5)` 等同於 `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`。

若要從整數範圍中選擇範例，請使用*range()* 物件作🄵引數。這對於從大型母體中取樣特🄵快速且節省空間：`sample(range(10000000), k=60)`。

如果樣本大小大於母體大小，*ValueError* 會被引發。

在 3.9 版的變更: 新增 *counts* 參數。

在 3.11 版的變更: *population* 必須是一個序列。不再支援將 set 自動轉🄵🄵 list。

9.6.5 離散分布

以下函式🄵生離散分🄵。

`random.`**`binomialvariate`**(*n=1*, *p=0.5*)

二項分🄵 (Binomial distribution)。回傳 *n* 個獨立試驗的成功次數，每個試驗的成功機率🄵 *p*:

數學上等價於:

```
sum(random() < p for i in range(n))
```

試驗次數 *n* 應🄵非負整數。成功的機率 *p* 應在 `0.0 <= p <= 1.0` 之間。結果是 `0 <= X <= n` 範圍🄵的整數。

在 3.12 版被加入.

9.6.6 實數分布

以下函式🄵生特定的實數分🄵。函式參數以分🄵方程中的對應變數命名，如常見的數學實踐所示；這些方程式中的大多數都可以在任意統計文本中找到。

`random.`**`random`**()

回傳範圍 `0.0 <= X < 1.0` 中的下一個隨機浮點數

`random.`**`uniform`**(*a*, *b*)

回傳一個隨機浮點數 *N*，當 `a <= b` 時確保 N 🄵 `a <= N <= b`、`b < a` 時確保 N 🄵 `b <= N <= a`。

終點值 `b` 可能包含在範圍🄵，也可能不包含在範圍🄵，取🄵於運算式 `a + (b-a) * random()` 中的浮點舍入。

`random.`**`triangular`**(*low*, *high*, *mode*)

回傳一個隨機浮點數 *N*，使得 `low <= N <= high`，🄵在這些邊界之間具有指定的 *mode*。*low* 和 *high* 邊界預設🄵零和一。*mode* 引數預設🄵邊界之間的中點，從而給出對稱分🄵。

`random.`**`betavariate`**(*alpha*, *beta*)

Beta（貝它）分布。參數的條件🄵 `alpha > 0` 和 `beta > 0`。回傳值的範圍介於 0 和 1 之間。

random.**expovariate**(*lambd=1.0*)

指數分🄵。*lambd* 🄵 1.0 除以所需的平均數。它應該不🄵零。（該參數將被稱🄵"lambda"，但這是 Python 中的保留字）如果 *lambd* 🄵正，則回傳值的範圍從 0 到正無窮大；如果 *lambd* 🄵負，則回傳值的範圍從負無窮大到 0。

在 3.12 版的變更: 新增 `lambd` 的預設值。

random.**gammavariate**(*alpha*, *beta*)

Gamma（伽瑪）分🄵。（不是 Gamma 函式！）。形狀 (shape) 和比例 (scale) 參數 *alpha* 和 *beta* 必須具有正值。（根據呼叫習慣不同，部分來源會將'beta' 定義🄵比例的倒數）。

Probability distribution function（機率密度函式）是：

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu=0.0*, *sigma=1.0*)

常態分🄵，也稱🄵高斯分🄵。*mu* 是平均數，*sigma* 是標准差。這比下面定義的*normalvariate()* 函式快一點。

多執行緒須注意：當兩個執行緒同時呼叫此函式時，它們可能會收到相同的傳回值。這可以透過三種方式避免。1）讓每個執行緒使用隨機數🄵生器的不同實例。2）在所有呼叫周圍加鎖。3）使用較慢但執行緒安全的*normalvariate()* 函式代替。

在 3.11 版的變更: *mu* 和 *sigma* 現在有預設引數。

random.**lognormvariate**(*mu*, *sigma*)

對數常態分🄵。如果你取此分🄵的自然對數，你將獲得一個具有平均數 *mu* 和標准差 *sigma* 的常態分🄵。*mu* 可以🄵任何值，🄵且 *sigma* 必須大於零。

random.**normalvariate**(*mu=0.0*, *sigma=1.0*)

常態分🄵。*mu* 是平均數，*sigma* 是標准差。

在 3.11 版的變更: *mu* 和 *sigma* 現在有預設引數。

random.**vonmisesvariate**(*mu*, *kappa*)

mu 是平均角度，以 0 到 2**pi* 之間的弧度表示，*kappa* 是濃度參數，必須大於或等於零。如果 *kappa* 等於零，則此分🄵在 0 到 2**pi* 的範圍🄵將🄵小🄵均🄵的隨機角度。

random.**paretovariate**(*alpha*)

Pareto distribution（柏拉圖分🄵）。*alpha* 是形狀參數。

random.**weibullvariate**(*alpha*, *beta*)

Weibull distribution（韋伯分🄵）。*alpha* 是比例參數，*beta* 是形狀參數。

9.6.7 替代🄵生器

class random.**Random**([*seed*])

實現*random* 模組使用的預設🄵隨機數🄵生器的 class。

在 3.11 版的變更: 過去 *seed* 可以是任何可雜🄵物件，但現在必須是以下類型之一：`None`、*int*、*float*、*str*、*bytes*、*bytearray*。

如果 `Random` 的子類🄵希望使用不同的基礎🄵生器，則應該覆寫以下方法：

seed(*a=None*, *version=2*)

在子類🄵中覆寫此方法以自訂 `Random` 實例的*seed()* 行🄵。

getstate()

在子類🄵中覆寫此方法以自訂 `Random` 實例的*getstate()* 行🄵。

setstate(*state*)

在子類🄵中覆寫此方法以自訂 Random 實例的*setstate()* 行🄵。

random()

在子類🄵中覆寫此方法以自訂 Random 實例的*random()* 行🄵。

或者，自訂🄵生器子類🄵還可以提供以下方法：

getrandbits(*k*)

在子類🄵中覆寫此方法以自訂 Random 實例的*getrandbits()* 行🄵。

randbytes(*n*)

在子類🄵中覆寫此方法以自訂 Random 實例的*randbytes()* 行🄵。

class random.**SystemRandom**([*seed*])

使用*os.urandom()* 函式從作業系統提供的來源🄵生隨機數的 Class。🄵非在所有系統上都可用。不依賴於軟體狀態，🄵且序列不可復現。因此*seed()* 方法🄵有效果且被忽略。如果呼叫*getstate()* 和*setstate()* 方法會引發*NotImplementedError*。

9.6.8 關於 Reproducibility（複現性）的注意事項

有時，能🄵重現🄵隨機數🄵生器給出的序列很有用。只要多執行緒未運行，透過重🄵使用種子值，同一序列就應該可以被復現。

大多數隨機 module 的演算法和 seed 設定函式在 Python 版本中可能會發生變化，但可以保證兩個方面不會改變：

- 如果增加了新的 seed 設定函式，則將提供向後相容的播種器 (seeder)。
- 當相容的播種器被賦予相同的種子時，🄵生器的*random()* 方法將持續🄵生相同的序列。

9.6.9 范例

基礎範例：

```
>>> random()                            # 隨機浮點數: 0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # 隨機浮點數: 2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # 到達間隔平均🄵 5 秒
5.148957571865031

>>> randrange(10)                       # 0 到 9（含）的整數
7\n

>>> randrange(0, 101, 2)                # 0 到 100（含）的偶數整數
26\n

>>> choice(['win', 'lose', 'draw'])     # 從序列中隨機選取單一元素
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # 打亂串列
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # 四個無重置的樣本
[40, 10, 50, 30]
```

模擬：

```
>>> # 六次輪盤旋轉（有重置的加權抽樣）
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # 從一副 52 張撲克牌中無重置發 20 張牌
>>> # 🄵確定具有十點值的牌的比例：十、🄵克、王后或國王
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # 估計從 7 次旋轉中得到 5 次或更多正面的概率
>>> # 使用一枚 60% 概率🄵正面的偏幣
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # 計算 5 個樣本的中位數位於中間兩個四分位數的概率
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

統計 bootstrapping（自助法）的範例，使用有重置的重新取樣來估計樣本平均數的信賴區間：

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

重新取樣排列測試的範例，來確定觀察到的藥物與安慰劑之間差🄵的統計學意義或 p 值：

```
# 範例來自 Dennis Shasha 和 Manda Wilson 的 "Statistics is Easy"
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} 次標🄵重新洗牌僅🄵生 {count} 個差🄵實例')
print(f'至少與觀察到的差🄵 {observed_diff:.1f} 一樣極端。')
print(f'單邊 p 值 {count / n:.4f} 使我們拒🄵無效假設')
print(f'即藥物與安慰劑之間🄵有差🄵。')
```

模擬多伺服器🄵列 (queue) 的到達時間與服務交付：

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles
```

(繼續下一頁)

(繼續上一頁)

```
average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # 每個伺服器可用的時間
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'平均等待時間：{mean(waits):.1f}   最大等待時間：{max(waits):.1f}')
print('四分位數：', [round(q, 1) for q in quantiles(waits)])
```

也參考

Statistics for Hackers 是由 Jake Vanderplas 🄵作的教學影片，僅使用幾個基本概念（包括模擬、取樣、洗牌、交叉驗證）進行統計分析。

Economics Simulation是由 Peter Norvig 對市場進行的模擬，顯示了該模組提供的許多工具和分🄵（高斯、均🄵、樣本、beta 變數、選擇，三角形、隨機數）的有效使用。

機率的具體介紹（使用 Python）🄵 Peter Norvig 的教學課程，涵蓋了機率理論的基礎知識與如何模擬以及使用 Python 執行數據分析。

9.6.10 使用方案

這些使用方案展示了如何有效地從*itertools*模組的組合🄵代器 (combinatoric iterators) 中進行隨機選擇：

```
def random_product(*args, repeat=1):
    "從 itertools.product(*args, **kwds) 中隨機選擇"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "從 itertools.permutations(iterable, r) 中隨機選擇"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "從 itertools.combinations(iterable, r) 中隨機選擇"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "有重置地選擇 r 個元素。結果按可🄵代物件排序。"
    # 結果將在 set(itertools.combinations_with_replacement(iterable, r)) 中
    pool = tuple(iterable)
    n = len(pool)
```

(繼續下一頁)

(繼續上一頁)

```
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

預設的*random()* 回傳 *0.0 ≤ x < 1.0* 範圍🄵 2^{-53} 的倍數。所有數字都是均🄵分🄵的，🄵且可以完全表示🄵 Python float。但是，該間隔中的許多其他可表示的 float 不是可能的選擇。例如 `0.05954861408025609` 不是 2^{-53} 的整數倍。

以下範例🄵用不同的方法。間隔中的所有 float 都是可能的選擇。尾數來自 $2^{52} \le$ *尾數* $< 2^{53}$ 範圍🄵的整數均🄵分🄵。指數來自幾何分🄵，其中小於 *-53* 的指數的出現頻率是下一個較大指數的一半。

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

Class 中的所有實數分🄵都將使用新方法：

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

該範例在概念上等效於一種演算法，該演算法從 *0.0 ≤ x < 1.0* 範圍🄵 2^{-1074} 的所有倍數中進行選擇。這些數字都是均🄵分🄵的，但大多數必須向下舍入到最接近的可表示的 Python float。（2^{-1074} 是最小🄵正的非正規化 float，等於 `math.ulp(0.0)`）

也參考

🄵生🄵隨機浮點值 Allen B. Downey 的一篇論文描述了🄵生比通常由*random()* 🄵生的 float 更 fine-grained（細粒的）的方法。

9.6.11 命令列用法

在 3.13 版被加入.

`random` 模組可以從命令列執行。

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

接受以下選項：

-h, --help

顯示幫助訊息🄵退出。

-c `CHOICE [CHOICE ...]`

--choice `CHOICE [CHOICE ...]`

列印一個隨機選擇，使用*choice()*。

-i `<N>`

--integer <N>
　列印 1 到 N（含）之間的隨機整數，使用*randint()*。

-f <N>

--float <N>
　列印 0 到 N（含）之間的隨機浮點數，使用*uniform()*。

如果未給定選項，則輸出取🄵於輸入：

- 字串或多個：與*--choice* 相同。
- 整數：與*--integer* 相同。
- 浮點數：與*--float* 相同。

9.6.12 命令列范例

以下是 random 命令列介面的一些範例：

```
$ # 隨機選擇一個
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a Mornay␣
↪sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # 隨機整數
$ python -m random 6
6

$ # 隨機浮點數
$ python -m random 1.8
1.7080016272295635

$ # 使用明確的引數
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux crevettes with a␣
↪Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

9.7 statistics --- 數學統計函式

在 3.4 版被加入.

原始碼： Lib/statistics.py

這個模組提供計算數值 (Real-valued) 資料的數學統計函式。

這個模組🄵非旨在與 NumPy、SciPy 等第三方函式庫，或者像 Minitab、SAS 和 Matlab 等專門設計給專業統計學家的高階統計軟體互相競🄵。此模組的目標在於繪圖和科學計算。

除非特🄵🄵明，這些函數支援*int*、*float*、*Decimal* 以及*Fraction*。目前不支援其他型🄵（無論是否🄵數值型🄵）。含有混合型🄵的資料的集合亦是尚未定義，且取🄵於該型🄵的實作。若你的輸入資料含有混合型🄵，你可以考慮使用*map()* 來確保結果是一致的，例如：`map(float, input_data)`。

有些資料集使用 `NaN`（非數）來表示缺漏的資料。由於 NaN 具有特殊的比較語義，在排序資料或是統計出現次數的統計函數中，會引發意料之外或是未定義的行🄵。受影響的函數包含 `median()`、`median_low()`、`median_high()`、`median_grouped()`、`mode()`、`multimode()` 以及 `quantiles()`。在呼叫這些函數之前，應該先移除 NaN 值：

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # This has surprising behavior
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # This result is unexpected
16.35

>>> sum(map(isnan, data))     # Number of missing values
2
>>> clean = list(filterfalse(isnan, data))  # Strip NaN values
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # Sorting now works as expected
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # This result is now well defined
18.75
```

9.7.1 平均值與中央位置量數

這些函式計算來自一個母體或樣本的平均值或代表值。

mean()	資料的算術平均數（平均值）。
fmean()	快速浮點數算數平均數，可調整權重。
geometric_mean()	資料的幾何平均數。
harmonic_mean()	資料的調和平均數。
kde()	Estimate the probability density distribution of the data.
kde_random()	Random sampling from the PDF generated by kde().
median()	資料的中位數（中間值）。
median_low()	資料中較小的中位數。
median_high()	資料中較大的中位數。
median_grouped()	分組資料的中位數（第 50 百分位數）。
mode()	離散 (discrete) 或名目 (nomial) 資料中的🄵數（出現次數最多次的值），只回傳一個。
multimode()	離散或名目資料中的🄵數（出現次數最多次的值）組成的 list。
quantiles()	將資料分成數個具有相等機率的區間，即分位數 (quantile)。

9.7.2 離度 (spread) 的測量

這些函式計算母體或樣本偏離平均值的程度。

pstdev()	資料的母體標准差。
pvariance()	資料的母體變🄵數。
stdev()	資料的樣本標准差。
variance()	資料的樣本變🄵數。

9.7.3 兩個輸入之間的關🄵統計

這些函式計算兩個輸入之間的關🄴統計數據。

covariance()	兩變數的樣本共變🄵數。
correlation()	Pearson 與 Spearman 相關🄴數 (correlation coefficient)。
linear_regression()	簡單🄵性🄵歸的斜率和截距。

9.7.4 函式細節

🄵：這些函數🄴不要求輸入的資料必須排序過。🄵了🄵讀方便，大部份的範例仍已排序過。

`statistics.`**`mean`**(*data*)

回傳 *data* 的樣本算數平均數，輸入可🄵一個 sequence 或者 iterable。

算數平均數🄴資料總和除以資料點的數目。他通常被稱🄴「平均值」，🄵管它只是🄴多不同的數學平均值之一。它是衡量資料集中位置的一種指標。

若 *data* 🄴空，則會引發*StatisticsError*。

使用範例：

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

備🄵

平均值🄴烈受到離群值 (outliers) 的影響，且不一定能當作這些資料點的典型範例。若要使用更穩健但效率較低的集中趨勢 (central tendency) 度量，請參考*median()*。

樣本平均數提供了對真實母體平均數的不偏估計 (unbiased estimate)，所以從所有可能的樣本中取平均值時，`mean(sample)` 會收斂至整個母體的真實平均值。若 *data* 🄵整個母體而非單一樣本，則 `mean(data)` 等同於計算真實的母體平均數 μ。

`statistics.`**`fmean`**(*data*, *weights=None*)

將 *data* 轉🄵🄵浮點數🄴計算其算數平均數。

這個函式運算比*mean()* 更快，🄵且它總是回傳一個*float*。*data* 可以是一個 sequence 或者 iterable。如果輸入的資料🄵空，則引發*StatisticsError*。

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

支援選擇性的加權。例如，一位教授以 20% 的比重計算小考分數，20% 的比重計算作業分數，30% 的比重計算期中考試分數，以及 30% 的比重計算期末考試分數：

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

如果有提供 *weights*，它必須與 *data* 長度相同，否則將引發*ValueError*。

在 3.8 版被加入.

在 3.11 版的變更: 新增 *weights* 的支援。

statistics.**geometric_mean**(*data*)

將 *data* 轉🄵成浮點數🄵計算其幾何平均數。

幾何平均數使用數值的乘積（與之對照，算數平均數使用的是數值的和）來表示 *data* 的集中趨勢或典型值。

若輸入的資料集🄵空、包含零、包含負值，則引發*StatisticsError*。*data* 可🄵 sequence 或者 iterable。

目前🄵有特🄵🄵了精確結果而特🄵多下什🄵工夫。（然而，未來或許會有。）

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

在 3.8 版被加入.

statistics.**harmonic_mean**(*data*, *weights=None*)

回傳 *data* 的調和平均數。*data* 可🄵實數 (real-valued) sequence 或者 iterable。如果省略 *weights* 或者 *weights* 🄵 `None`，則假設各權重相等。

調和平均數是資料的倒數 (reciprocal) 經過*mean()* 運算過後的倒數。例如，三個數 *a*，*b* 與 *c* 的調和平均數等於 `3/(1/a + 1/b + 1/c)`。若其中一個值🄵零，結果將🄵零。

調和平均數是一種平均數，是衡量資料中心位置的一種方法。它通常用於計算比率 (ratio) 或率 (rate) 的平均，例如速率（speed）。

假設一輛汽車以時速 40 公里的速率行駛 10 公里，然後再以時速 60 公里的速率行駛 10 公里，求汽車的平均速率?

```
>>> harmonic_mean([40, 60])
48.0
```

假設一輛汽車以時速 40 公里的速率行駛 5 公里，然後在交通順暢時，加速到時速 60 公里，以此速度行駛剩下的 30 公里。求汽車的平均速率?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

若 *data* 🄵空、含有任何小於零的元素、或者加權總和不🄵正數，則引發*StatisticsError*。

目前的演算法設計🄵，若在輸入當中遇到零，則會提前退出。這意味著後續的輸入🄵未進行有效性檢查。（這種行🄵在未來可能會改變。）

在 3.6 版被加入.

在 3.10 版的變更: 新增 *weights* 的支援。

statistics.**kde**(*data*, *h*, *kernel='normal'*, *, *cumulative=False*)

Kernel Density Estimation (KDE): Create a continuous probability density function or cumulative distribution function from discrete samples.

The basic idea is to smooth the data using a kernel function. to help draw inferences about a population from a sample.

The degree of smoothing is controlled by the scaling parameter *h* which is called the bandwidth. Smaller values emphasize local features while larger values give smoother results.

The *kernel* determines the relative weights of the sample data points. Generally, the choice of kernel shape does not matter as much as the more influential bandwidth smoothing parameter.

Kernels that give some weight to every sample point include *normal* (*gauss*), *logistic*, and *sigmoid*.

Kernels that only give weight to sample points within the bandwidth include *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight*, and *cosine*.

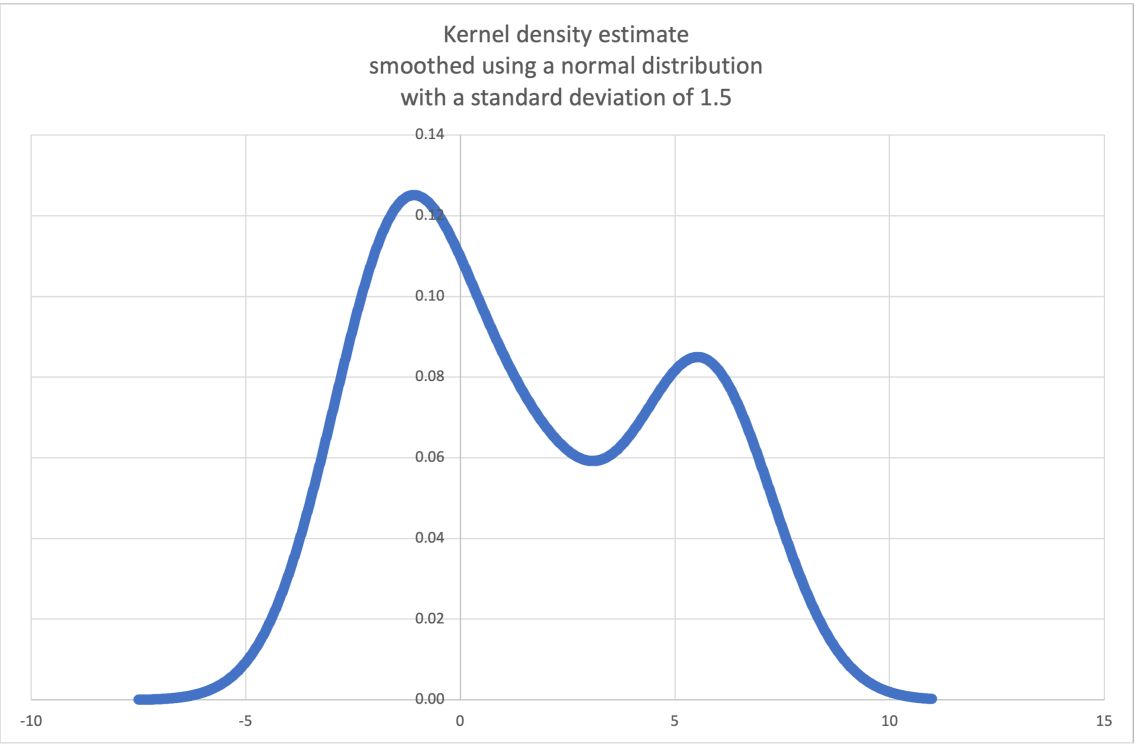
If *cumulative* is true, will return a cumulative distribution function.

若 *data* 🄵空，則引發*StatisticsError*。

維基百科有一個範例，我們可以使用*kde()* 來從一個小樣本生成🄵繪🄵估計的機率密度函數：

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

`xarr` 和 `yarr` 中的點可用於繪🄵 PDF 圖：



在 3.13 版被加入.

`statistics.`**`kde_random`**(*data*, *h*, *kernel='normal'*, *, *seed=None*)

Return a function that makes a random selection from the estimated probability density function produced by `kde(data, h, kernel)`.

Providing a *seed* allows reproducible selections. In the future, the values may change slightly as more accurate kernel inverse CDF estimates are implemented. The seed may be an integer, float, str, or bytes.

若 *data* 🄵空，則引發*StatisticsError*。

Continuing the example for *kde()*, we can use *kde_random()* to generate new random selections from an estimated probability density function:

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

在 3.13 版被加入.

statistics.**median**(*data*)

使用常見的「中間兩數取平均」方法回傳數值資料的中位數（中間值）。若 *data* 🄵空，則會引發*StatisticsError*。*data* 可🄵一個 sequence 或者 iterable。

中位數是一種穩健的衡量資料中心位置的方法，較不易被離群值影響。當資料點數量🄵奇數時，會回傳中間的資料點：

```
>>> median([1, 3, 5])
3
```

當資料點數量🄵偶數時，中位數透過中間兩個值的平均數來插值計算：

```
>>> median([1, 3, 5, 7])
4.0
```

若你的資料🄵離散資料，🄵且你不介意中位數可能🄵非真實的資料點，那這函式適合你。

若你的資料🄵順序 (ordinal) 資料（支援排序操作）但🄵非數值型（不支援加法），可以考慮改用*median_low()* 或是*median_high()* 代替。

statistics.**median_low**(*data*)

回傳數值型資料的低中位數 (low median)。若 *data* 🄵空，則引發*StatisticsError*。*data* 可🄵 sequence 或者 iterable。

低中位數一定會在原本的資料集當中。當資料點數量🄵奇數時，回傳中間值。當數量🄵偶數時，回傳兩個中間值當中較小的值。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

當你的資料🄵離散資料，且你希望中位數是實際的資料點而不是插值時，可以用低中位數。

statistics.**median_high**(*data*)

回傳數值型資料的高中位數 (high median)。若 *data* 🄵空，則引發*StatisticsError*。*data* 可🄵 sequence 或者 iterable。

高中位數一定會在原本的資料集當中。當資料點數量🄵奇數時，回傳中間值。當數量🄵偶數時，回傳兩個中間值當中較大的值。

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

當你的資料🄵離散資料，且你希望中位數是實際的資料點而不是插值時，可以用高中位數。

statistics.**median_grouped**(*data*, *interval=1.0*)

Estimates the median for numeric data that has been grouped or binned around the midpoints of consecutive, fixed-width intervals.

The *data* can be any iterable of numeric data with each value being exactly the midpoint of a bin. At least one value must be present.

The *interval* is the width of each bin.

For example, demographic information may have been summarized into consecutive ten-year age groups with each group being represented by the 5-year midpoints of the intervals:

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 to 30 years old
...    35: 484,   # 30 to 40 years old
```

(繼續下一頁)

(繼續上一頁)

```
...     45: 387,   # 40 to 50 years old
...     55:  22,   # 50 to 60 years old
...     65:   6,   # 60 to 70 years old
... })
...
```

The 50th percentile (median) is the 536th person out of the 1071 member cohort. That person is in the 30 to 40 year old age group.

The regular *median()* function would assume that everyone in the tricenarian age group was exactly 35 years old. A more tenable assumption is that the 484 members of that age group are evenly distributed between 30 and 40. For that, we use *median_grouped()*:

```
>>> data = list(demographics.elements())
>>> median(data)
35
>>> round(median_grouped(data, interval=10), 1)
37.5
```

The caller is responsible for making sure the data points are separated by exact multiples of *interval*. This is essential for getting a correct result. The function does not check this precondition.

Inputs may be any numeric type that can be coerced to a float during the interpolation step.

statistics.**mode**(*data*)

回傳離散或名目 *data* 中出現次數最多次的值，只回傳一個。🄵數（如果存在）是最典型的值，🄵用來衡量資料的中心位置。

若有多個出現次數相同的🄵數，則回傳在 *data* 中最先出現的🄵數。如果希望回傳其中最小或最大的🄵數，可以使用 `min(multimode(data))` 或 `max(multimode(data))`。如果輸入的 *data* 🄵空，則會引發*StatisticsError*。

`mode` 假定🄵離散資料，🄵回傳單一的值。這也是一般學校教授的標準🄵數定義：

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

🄵數特🄵之處在於它是此套件中唯一也適用於名目（非數值型）資料的統計量：

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Only hashable inputs are supported. To handle type *set*, consider casting to *frozenset*. To handle type *list*, consider casting to *tuple*. For mixed or nested inputs, consider using this slower quadratic algorithm that only depends on equality tests: `max(data, key=data.count)`.

在 3.8 版的變更: 現在，遇到資料中有多個🄵數時，會回傳第一個遇到的🄵數。在以前，當找到大於一個🄵數時，會引發*StatisticsError*。

statistics.**multimode**(*data*)

回傳一個 list，其組成🄵 *data* 中出現次數最多次的值，🄵按照它們在 *data* 中首次出現的順序排列。如果有多個🄵數，將會回傳所有結果。若 *data* 🄵空，則回傳空的 list：

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

在 3.8 版被加入.

statistics.**pstdev**(*data*, *mu=None*)

回傳母體標准差（即母體變🄵數的平方根）。有關引數以及其他細節，請參見*pvariance()*。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

回傳 *data* 的母體變🄵數。*data* 可🄵非空實數 sequence 或者 iterable。變🄵數，或者以平均數🄵中心的二階動差，用於衡量資料的變🄵性（離度或分散程度）。變🄵數大表示資料分散，變🄵數小表示資料集中在平均數附近。

若有傳入選擇性的第二個引數 *mu*，該引數應該要是 *data* 的母體平均值 (population mean)。它也可以用於計算非以平均值🄵中心的第二動差。如果🄵有傳入此引數或者引數🄵 `None` （預設值），則自動計算資料的算數平均數。

使用此函式來計算整個母體的變🄵數。如果要從樣本估算變🄵數，*variance()* 通常是較好的選擇。

若 *data* 🄵空，則引發*StatisticsError*。

範例：

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

如果已經計算出資料的平均值，你可以將其作🄵選擇性的第二個引數 *mu* 傳遞以避免重新計算：

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

支援小數 (decimal) 與分數 (fraction)：

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

備🄵

當在整個母體上呼叫此函式時，會回傳母體變🄵數 σ^2。當在樣本上呼叫此函式時，會回傳有偏差的樣本變🄵數 s^2，也就是具有 N 個自由度的變🄵數。

若你以某種方式知道真正的母體平均數 μ，你可以將一個已知的母體平均數作🄵第二個引數提供給此函式，用以計算樣本的變🄵數。只要資料點是母體的隨機樣本，結果將是母體變🄵數的不偏估計。

statistics.**stdev**(*data*, *xbar=None*)

回傳樣本標准差（即樣本變🄵數的平方根）。有關引數以及其他細節，請參見*variance()*。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

回傳 *data* 的樣本變🄵數。*data* 🄵兩個值以上的實數 iterable。變🄵數，或者以平均數🄵中心的二階動差，用於衡量資料的變🄵性（離度或分散程度）。變🄵數大表示資料分散，變🄵數小表示資料集中在平均數附近。

若有傳入選擇性的第二個引數 *xbar*，它應該是 *data* 的樣本平均值 (sample mean)。如果🄵有傳入或者🄵 `None` （預設值），則自動計算資料的平均值。

當你的資料是來自母體的樣本時，請使用此函式。若要從整個母體計算變🄵數，請參見*pvariance()*。

若 *data* 🄵少於兩個值，則引發*StatisticsError*。

範例：

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

如果已經計算出資料的樣本平均值，你可以將其作🄵選擇性的第二個引數 *mu* 傳遞以避免重新計算：

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

此函式不會驗證你傳入的 *xbar* 是否🄵實際的平均數。傳入任意的 *xbar* 會導致無效或不可能的結果。

支援小數 (decimal) 與分數 (fraction)：

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

備🄵

這是經過 Bessel 校正 (Bessel's correction) 後的樣本變🄵數 s^2，又稱🄵自由度🄵 N-1 的變🄵數。只要資料點具有代表性（例如：獨立且具有相同分布），結果應該會是對真實母體變🄵數的不偏估計。

若你剛好知道真正的母體平均數 μ，你應該將其作🄵 *mu* 參數傳入*pvariance()* 函式來計算樣本變🄵數。

statistics.**quantiles**(*data*, *, *n=4*, *method='exclusive'*)

將 *data* 分成 *n* 個具有相等機率的連續區間。回傳一個包含 `n - 1` 個用於切分各區間的分隔點的 list。

將 *n* 設🄵 4 以表示四分位數 (quartile)（預設值）。將 *n* 設置🄵 100 表示百分位數 (percentile)，這將給出 99 個分隔點將 *data* 分成 100 個大小相等的組。如果 *n* 不是至少🄵 1，則引發*StatisticsError*。

data 可以是包含樣本資料的任何 iterable。🄵了取得有意義的結果，*data* 中的資料點數量應大於 *n*。如果連一個資料點都🄵有，則引發*StatisticsError*。

分隔點是從兩個最近的資料點🄵性🄵插值計算出來的。舉例來🄵，如果分隔點落在兩個樣本值 `100` 與 `112` 之間的距離三分之一處，則分隔點的值將🄵 `104`。

計算分位數的 *method* 可以根據 *data* 是否包含或排除來自母體的最小與最大可能的值而改變。

預設的 *method* 是"exclusive"，用於從可能找到比樣本更極端的值的母體中抽樣的樣本資料。對於 *m* 個已排序的資料點，計算出低於 *i-th* 的部分🄵 `i / (m + 1)`。給定九個樣本資料，此方法將對資料排序且計算下列百分位數：10%、20%、30%、40%、50%、60%、70%、80%、90%。

若將 *method* 設🄵"inclusive"，則用於描述母體或者已知包含母體中最極端值的樣本資料。在 *data* 中的最小值被視🄵第 0 百分位數，最大值🄵第 100 百分位數。對於 *m* 個已排序的資料點，計算出低於 *i-th* 的部分🄵 `(i - 1) / (m - 1)`。給定十一個個樣本資料，此方法將對資料排序且計算下列百分位數：0%、10%、20%、30%、40%、50%、60%、70%、80%、90%、100%。

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
```

(繼續下一頁)

(繼續上一頁)

```
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

在 3.8 版被加入.

在 3.13 版的變更: No longer raises an exception for an input with only a single data point. This allows quantile estimates to be built up one sample point at a time becoming gradually more refined with each new data point.

statistics.**covariance**(*x*, *y*, */*)

回傳兩輸入 *x* 與 *y* 的樣本共變🄵數 (sample covariance)。共變🄵數是衡量兩輸入的聯合變🄵性 (joint variability) 的指標。

兩輸入必須具有相同長度（至少兩個），否則會引發*StatisticsError*。

範例:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

在 3.10 版被加入.

statistics.**correlation**(*x*, *y*, */*, *, *method='linear'*)

回傳兩輸入的 Pearson 相關🄵數 (Pearson's correlation coefficient)。Pearson 相關🄵數 *r* 的值介於 -1 與 +1 之間。它衡量🄵性關🄵的🄵度與方向。

如果 *method* 🄵"ranked"，則計算兩輸入的 Spearman 等級相關🄵數 (Spearman's rank correlation coefficient)。資料將被取代🄵等級。平手的情🄵則取平均，令相同的值排名也相同。所得🄵數衡量單調關🄵 (monotonic relationship) 的🄵度。

Spearman 相關🄵數適用於順序型資料，或者不符合 Pearson 相關🄵數要求的🄵性比例關🄵的連續型 (continuous) 資料。

兩輸入必須具有相同長度（至少兩個），且不須🄵常數，否則會引發*StatisticsError*。

以 Kepler 行星運動定律🄵例:

```
>>> # Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and  Neptune
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756, 30_687, 60_190]    # days
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900, 4_500] # million km

>>> # Show that a perfect monotonic relationship exists
>>> correlation(orbital_period, dist_from_sun, method='ranked')
1.0

>>> # Observe that a linear relationship is imperfect
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # Demonstrate Kepler's third law: There is a linear correlation
>>> # between the square of the orbital period and the cube of the
>>> # distance from the sun.
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
```

(繼續下一頁)

(繼續上一頁)

```
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

在 3.10 版被加入.

在 3.12 版的變更: 新增了對 Spearman 等級相關🄵數的支援。

statistics.**linear_regression**(*x*, *y*, */*, ***, *proportional=False*)

回傳使用普通最小平方法 (ordinary least square) 估計出的簡單🄵性🄵歸 (simple linear regression) 參數中的斜率 (slope) 與截距 (intercept)。簡單🄵性🄵歸描述自變數 (independent variable) *x* 與應變數 (dependent variable) *y* 之間的關🄵，用以下的🄵性函式表示：

y = *slope* * *x* + *intercept* + *noise*

其中 `slope` 和 `intercept` 是被估計的🄵歸參數，而 `noise` 表示由🄵性🄵歸未解釋的資料變🄵性（它等於應變數的預測值與實際值之差）。

兩輸入必須具有相同長度（至少兩個），且自變數 *x* 不得🄵常數，否則會引發*StatisticsError*。

舉例來🄵，我們可以使用 Monty Python 系列電影的上映日期來預測至 2019 年🄵止，假設他們保持固定的🄵作速度，應該會🄵生的 Monty Python 電影的累計數量。

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

若將 *proportional* 設🄵 True，則假設自變數 *x* 與應變數 *y* 是直接成比例的，資料座落在通過原點的一直🄵上。由於 *intercept* 始終🄵 0.0，因此🄵性函式可簡化如下：

y = *slope* * *x* + *noise*

繼續*correlation()* 中的範例，我們看看基於主要行星的模型可以如何很好地預測矮行星的軌道距離：

```
>>> model = linear_regression(period_squared, dist_cubed, proportional=True)
>>> slope = model.slope

>>> # 矮行星:           冥王星、   🄵神星、   鳥神星、   妊神星、   🄵神星
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 1_680]  # 天數
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # 以百萬公里🄵單位的實際距離
[5906, 10152, 6796, 6450, 414]
```

在 3.10 版被加入.

在 3.11 版的變更: 新增 *proportional* 的支援。

9.7.5 例外

定義了一個單一的例外：

exception statistics.**StatisticsError**

ValueError 的子類🄵，用於和統計相關的例外。

9.7.6 NormalDist 物件

NormalDist 是一種用於建立與操作隨機變數 (random variable) 的常態分布的工具。它是一個將量測資料的平均數與標準差視🄵單一實體的類🄵。

常態分布源自於中央極限定理 (Central Limit Theorem)，在統計學中有著廣泛的應用。

class statistics.**NormalDist**(*mu=0.0*, *sigma=1.0*)

此方法會回傳一個新 *NormalDist* 物件，其中 *mu* 代表算數平均數而 *sigma* 代表標準差。

若 *sigma* 🄵負值，則引發 *StatisticsError*。

mean

常態分布中的算數平均數唯讀屬性。

median

常態分布中的中位數唯讀屬性。

mode

常態分布中的🄵數唯讀屬性。

stdev

常態分布中的標準差唯讀屬性。

variance

常態分布中的變🄵數唯讀屬性。

classmethod from_samples(*data*)

利用 *fmean()* 與 *stdev()* 函式，估計 *data* 的 *mu* 與 *sigma* 參數，建立一個常態分布的實例。

data 可以是任何*iterable*，🄵應包含可以轉🄵🄵 *float* 的值。若 *data* 🄵有包含至少兩個以上的元素在🄵，則引發 *StatisticsError*，因🄵至少需要一個點來估計中央值且至少需要兩個點來估計分散情形。

samples(*n*, *, *seed=None*)

給定平均值與標準差，🄵生 *n* 個隨機樣本。回傳一個由 *float* 組成的 *list*。

若有給定 *seed*，則會建立一個以此🄵基礎的亂數🄵生器實例。這對於建立可重現的結果很有幫助，即使在多執行緒情境下也是如此。

在 3.13 版的變更.

Switched to a faster algorithm. To reproduce samples from previous versions, use *random.seed()* and *random.gauss()*.

pdf(*x*)

利用機率密度函數 (probability density function, pdf) 計算隨機變數 *X* 接近給定值 *x* 的相對概度 (relative likelihood)。數學上，它是比率 `P(x <= X < x+dx) / dx` 在 *dx* 趨近於零時的極限值。

相對概度是樣本出現在狹窄範圍的機率，除以該範圍的寬度（故稱🄵「密度」）計算而得。由於概度是相對於其它點，故其值可大於 `1.0`。

cdf(*x*)

利用累積分布函式 (cumulative distribution function, cdf) 計算隨機變數 *X* 小於或等於 *x* 的機率。數學上，它記🄵 `P(X <= x)`。

inv_cdf(*p*)

計算反累計分布函式 (inverse cumulative distribution function)，也稱🄵分位數函式 (quantile function) 或者百分率點 (percent-point) 函式。數學上記🄵 `x : P(X <= x) = p`。

找出一個值 *x*，使得隨機變數 *X* 小於或等於該值的機率等於給定的機率 *p*。

overlap(*other*)

衡量兩常態分布之間的一致性。回傳一個介於 0.0 與 1.0 之間的值，表示兩機率密度函式的重🄵區域。

quantiles(*n=4*)

將常態分布分割成 *n* 個具有相等機率的連續區間。回傳一個 list，包含 (n-1) 個切割區間的分隔點。

將 *n* 設定🄵 4 表示四分位數（預設值）。將 *n* 設定🄵 10 表示十分位數。將 *n* 設定🄵 100 表示百分位數，這會🄵生 99 個分隔點，將常態分布切割成大小相等的群組。

zscore(*x*)

計算標准分數 (Standard Score)，用以描述在常態分布中，*x* 高出或低於平均數幾個標准差：`(x - mean) / stdev`。

在 3.9 版被加入.

NormalDist 的實例支援對常數的加法、🄵法、乘法與除法。這些操作用於平移與縮放。例如：

```
>>> temperature_february = NormalDist(5, 2.5)             # 攝氏
>>> temperature_february * (9/5) + 32                     # 華氏
NormalDist(mu=41.0, sigma=4.5)
```

不支援將常數除以*NormalDist* 的實例，因🄵結果將不符合常態分布。

由於常態分布源自於自變數的加法效應 (additive effects)，因此可以將兩個獨立的常態分布隨機變數相加與相🄵，🄵且表示🄵*NormalDist* 的實例。例如：

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

在 3.8 版被加入.

9.7.7 范例與錦囊妙計

經典機率問題

NormalDist 可以輕易地解🄵經典的機率問題。

例如，給定 SAT 測驗的歷史資料，顯示成績🄵平均 1060、標准差 195 的常態分布。我們要求出分數在 1100 與 1200 之間（四舍五入至最接近的整數）的學生的百分比：

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

找出 SAT 分數的四分位數以及十分位數：

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

用於模擬的蒙地卡羅 (Monte Carlo) 輸入

欲估計一個不易透過解析方法求解的模型的分布，*NormalDist* 可以🄵生輸入樣本以進行蒙地卡羅模擬：

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
```

(繼續下一頁)

(繼續上一頁)

```
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

近似二項分布

當樣本數量🄵大，且試驗成功的機率接近 50%，可以使用常態分布來近似二項分布 (Binomial distributions)。

例如，一場有 750 位參加者的開源研討會中，有兩間可容納 500 人的會議室。一場是關於 Python 的講座，另一場則是關於 Ruby 的。在過去的會議中，有 65% 的參加者傾向參與 Python 講座。假設參與者的偏好🄵有改變，那🄵 Python 會議室未超過自身容量限制的機率是?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Exact solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

單純貝氏分類器 (Naive bayesian classifier)

常態分布常在機器學習問題中出現。

維基百科有個 Naive Bayesian Classifier 的優良範例。課題🄵從身高、體重與鞋子尺寸等符合常態分布的特徵量測值中判斷一個人的性🄵。

給定一組包含八個人的量測值的訓練資料集。假設這些量測值服從常態分布，我們可以利用*NormalDist*來總結資料:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

接著，我們遇到一個新的人，他的特徵量測值已知，但性🄵未知:

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
>>> fs = 8          # foot size
```

從可能🄵男性或女性的 50% 先驗機率 (prior probability) 🄵開端，我們將後驗機率 (posterior probability) 計算🄵先驗機率乘以給定性🄵下，各特徵量測值的概度乘積:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

最終的預測結果將取🄵於最大的後驗機率。這被稱🄵最大後驗機率 (maximum a posteriori) 或者 MAP：

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPTER 10

函式程式設計模組

本章節所描述的模組提供了支援函式程式設計風格的函式和類🄵，以及對可呼叫物件的一般操作。

本章包含下列的模組：

10.1 itertools --- 建立🄵生高效率🄵圈之🄵代器的函式

這個模組實作了許多🄵代器 *(iterator)* 構建塊 (building block)，其靈感來自 APL、Haskell 和 SML 的結構。每個構建塊都以適合 Python 的形式來重新設計。

這個模組標准化了快速且高效率利用記憶體的核心工具集，這些工具本身或組合使用都很有用。它們共同構成了一個「🄵代器代數 (iterator algebra)」，使得在純 Python 中簡潔且高效地建構專用工具成🄵可能。

例如，SML 提供了一個造表工具：`tabulate(f)`，它🄵生一個序列 `f(0), f(1), ...`。在 Python 中，可以透過結合*`map()`* 和*`count()`* 組成 `map(f, count())` 以達到同樣的效果。

無限🄵代器：

🄵代器	引數	結果	範例
`count()`	[start[, step]]	start, start+step, start+2*step, ...	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem，... 重🄵無限次或 n 次	`repeat(10, 3) → 10 10 10`

在最短輸入序列 (shortest input sequence) 處終止的🄵代器：

🄵代器	引數	結果	範例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
batched()	p, n	(p0, p1, ..., p_n-1), ...	`batched('ABCDEFG', n=2) → AB CD EF G`
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') → A B C D E F`
chain.from_iterable()	可🄵代物件	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1, 1]) → A C E F`
dropwhile()	predicate, seq	seq[n], seq[n+1]，當 predicate 失敗時開始	`dropwhile(lambda x: x<5, [1,4,6, 3,8]) → 6 3 8`
filterfalse()	predicate, seq	當 predicate(elem) 失敗時 seq 的元素	`filterfalse(lambda x: x<5, [1,4, 6,3,8]) → 6 8`
groupby()	iterable[, key]	根據 key(v) 的值分組的子🄵代器	`groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)`
islice()	seq, [start,] stop [, step]	seq[start:stop:step] 的元素	`islice('ABCDEFG', 2, None) → C D E F G`
pairwise()	可🄵代物件	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') → AB BC CD DE EF FG`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10, 3)]) → 32 9 1000`
takewhile()	predicate, seq	seq[0], seq[1]，直到 predicate 失敗	`takewhile(lambda x: x<5, [1,4,6, 3,8]) → 1 4`
tee()	it, n	it1, it2, ... itn，將一個🄵代器分成 n 個	`tee('ABC', 2) → A B C, A B C`
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-`

組合🄵代器：

🄵代器	引數	結果
product()	p, q, ... [repeat=1]	笛卡爾乘積 (cartesian product)，相當於巢狀的 for 🄵圈
permutations()	p[, r]	長度🄵 r 的元組，所有可能的定序，無重🄵元素
combinations()	p, r	長度🄵 r 的元組，按照排序過後的定序，無重🄵元素
combinations_with_replacement()	p, r	長度🄵 r 的元組，按照排序過後的定序，有重🄵元素

範例	結果
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool 函式

以下的函式都會建構🄴回傳🄵代器。一些函式提供無限長度的串流 (stream)，因此應僅由截斷串流的函式或🄵圈來存取它們。

itertools.**accumulate**(*iterable*[, *function*, *, *initial=None*])

建立一個回傳累積和的🄵代器，或其他二進位函式的累積結果。

function 預設🄴加法。*function* 應接受兩個引數，即累積總和和來自 *iterable* 的值。

如果提供了 *initial* 值，則累積將從該值開始，🄵且輸出的元素數將比輸入的可🄵代物件多一個。

大致等價於：

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

function 引數可以被設定🄵*min()* 以得到連續的最小值，設定🄵*max()* 以得到連續的最大值，或者設定🄵*operator.mul()* 以得到連續的乘積。也可以透過累積利息和付款來建立攤銷表 (Amortization tables)：

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))                # 運行最大值
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))       # 運行乘積
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# 攤銷一筆 1000 的 5% 貸款，分 10 年、每年支付 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

可參見*functools.reduce()*，其是個類似的函式，但僅回傳最終的累積值。

在 3.2 版被加入.

在 3.3 版的變更: 新增可選的 *function* 參數。

在 3.8 版的變更: 新增可選的 *initial* 參數。

itertools.**batched**(*iterable*, *n*, *, *strict=False*)

將來自 *iterable* 的資料分批🄵長度🄵 *n* 的元組。最後一個批次可能比 *n* 短。

如果 *strict* 🄵真，則當最後一個批次比 *n* 短時，會引發*ValueError*。

對輸入的可🄴代物件進行🄵圈，🄵將資料累積到大小🄵 *n* 的元組中。輸入是惰性地被消耗 (consumed lazily) 的，會剛好足🄵填充一批的資料。一旦批次填滿或輸入的可🄴代物件耗盡，就會 yield 出結果：

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

大致等價於：

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 2) → AB CD EF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

在 3.12 版被加入.

在 3.13 版的變更: 新增 *strict* 選項。

itertools.**chain**(**iterables*)

建立一個🄵代器，從第一個可🄵代物件回傳元素直到其耗盡，然後繼續處理下一個可🄵代物件，直到所有可🄵代物件都耗盡。這將多個資料來源結合🄵單一個🄵代器。大致等價於：

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.**from_iterable**(*iterable*)

chain() 的另一個建構函式。從單個可🄵代的引數中得到鏈接的輸入，該引數是惰性計算的。大致等價於：

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

從輸入 *iterable* 中回傳長度🄵 *r* 的元素的子序列。

輸出是*product()* 的子序列，僅保留作🄵 *iterable* 子序列的條目。輸出的長度由*math.comb()* 給定，當 `0 ≤ r ≤ n` 時，長度🄵 `n! / r! / (n - r)!`，當 `r > n` 時🄵零。

根據輸入值 *iterable* 的順序，組合的元組會按照字典順序輸出。如果輸入的 *iterable* 已經排序，則輸出的元組也將按排序的順序🄵生。

元素是根據它們的位置（而非值）來🄵定其唯一性。如果輸入的元素都是獨特的，則每個組合🄵將不會有重🄵的值。

大致等價於：

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123

    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
```

(繼續下一頁)

(繼續上一頁)

```
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.**combinations_with_replacement**(*iterable*, *r*)

回傳來自輸入 *iterable* 的長度🄵 *r* 的子序列，且允許個🄵元素重🄵多次。

其輸出是一個*product()* 的子序列，僅保留作🄵 *iterable* 子序列（可能有重🄵元素）的條目。當 `n > 0` 時，回傳的子序列數量🄵 `(n + r - 1)! / r! / (n - 1)!`。

根據輸入值 *iterable* 的順序，組合的元組會按照字典順序輸出。如果輸入的 *iterable* 已經排序，則輸出的元組也將按排序的順序🄵生。

元素是根據它們的位置（而非值）來🄴定其唯一性。如果輸入的元素都是獨特的，生成的組合也將是獨特的。

大致等價於：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) → AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

在 3.1 版被加入.

itertools.**compress**(*data*, *selectors*)

建立一個🄵代器，回傳 *data* 中對應 *selectors* 的元素🄵 true 的元素。當 *data* 或 *selectors* 可🄵代物件耗盡時停止。大致等價於：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

在 3.1 版被加入.

itertools.**count**(*start=0*, *step=1*)

建立一個🄵代器，回傳從 *start* 開始的等差的值。可以與*map()* 一起使用來🄵生連續的資料點，或與*zip()* 一起使用來增加序列號。大致等價於：

```
def count(start=0, step=1):
    # count(10) → 10 11 12 13 14 ...
    # count(2.5, 0.5) → 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

當用浮點數計數時，將上述程式碼替🄵🄵乘法有時可以獲得更好的精確度，例如：`(start + step * i for i in count())`。

在 3.1 版的變更: 新增 *step* 引數🄴允許非整數引數。

itertools.**cycle**(*iterable*)

建立一個🄴代器，回傳 *iterable* 中的元素🄴保存每個元素的副本。當可🄴代物件耗盡時，從保存的副本中回傳元素。會無限次的重🄴。大致等價於：

```
def cycle(iterable):
    # cycle('ABCD') → A B C D A B C D A B C D ...

    saved = []
    for element in iterable:
        yield element
        saved.append(element)

    while saved:
        for element in saved:
            yield element
```

此 itertool 可能需要大量的輔助儲存空間（取🄴於可🄴代物件的長度）。

itertools.**dropwhile**(*predicate*, *iterable*)

建立一個🄴代器，在 *predicate* 🄵 true 時🄵🄵 *iterable* 中的元素，之後回傳每個元素。大致等價於：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

注意，在 predicate 首次變🄵 False 之前，這不會🄴生任何輸出，所以此 itertool 可能會有較長的🄴動時間。

itertools.**filterfalse**(*predicate*, *iterable*)

建立一個🄴代器，過濾 *iterable* 中的元素，僅回傳 *predicate* 🄵 False 值的元素。如果 *predicate* 是 `None`，則回傳🄵 False 的項目。大致等價於：

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8

    if predicate is None:
        predicate = bool

    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

建立一個🄴代器，回傳 *iterable* 中連續的鍵和群組。*key* 是一個🄵每個元素計算鍵值的函式。如果其未指定或🄵 None，則 *key* 預設🄵一個識🄴性函式 (identity function)，🄴回傳未被更改的元素。一般來🄴，可🄴代物件需要已經用相同的鍵函式進行排序。

groupby() 的操作類似於 Unix 中的 uniq 過濾器。每當鍵函式的值發生變化時，它會🄴生一個 break 或新的群組（這就是🄴什🄴通常需要使用相同的鍵函式對資料進行排序）。這種行🄴不同於 SQL 的 GROUP BY，其無論輸入順序如何都會聚合相同的元素。

回傳的群組本身是一個與*groupby()* 共享底層可🄴代物件的🄴代器。由於來源是共享的，當*groupby()* 物件前進時，前一個群組將不再可見。因此，如果之後需要該資料，應將其儲存🄵串列：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # 將群組🄵代器儲存🄵串列
    uniquekeys.append(k)
```

groupby() 大致等價於：

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] → A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] → AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
    except StopIteration:
        return
    curr_key = keyfunc(curr_value)

    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

建立一個🄴代器，回傳從 iterable 中選取的元素。其作用類似於序列切片 (sequence slicing)，但不支援負數的 *start*、*stop* 或 *step* 的值。

如果 *start* 🄵零或 None，則從零開始🄴代。否則在達到 *start* 之前，會跳過 iterable 中的元素。

如果 *stop* 🄵 None，則🄴代將繼續前進直到輸入耗盡。如果指定了 *stop*，則在達到指定位置時停止。

如果 *step* 🄵 None，則步長 (step) 預設🄵一。元素會連續回傳，除非將 *step* 設定🄵大於一，這會導致一些項目被跳過。

大致等價於：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) → A B
    # islice('ABCDEFG', 2, 4) → C D
    # islice('ABCDEFG', 2, None) → C D E F G
    # islice('ABCDEFG', 0, None, 2) → A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

若輸入🄵🄵代器，則完整耗盡 *islice* 會使輸入的🄵代器向前移動 `max(start, stop)` 步，與 *step* 的值無關。

itertools.**pairwise**(*iterable*)

回傳從輸入的 *iterable* 中提取的連續重🄵對。

輸出🄵代器中的 2 元組數量將比輸入少一個。如果輸入的可🄵代物件中的值少於兩個，則輸出將🄵空值。

大致等價於：

```
def pairwise(iterable):
    # pairwise('ABCDEFG') → AB BC CD DE EF FG

    iterator = iter(iterable)
    a = next(iterator, None)

    for b in iterator:
        yield a, b
        a = b
```

在 3.10 版被加入.

itertools.**permutations**(*iterable*, *r=None*)

回傳 *iterable* 中連續且長度🄵 *r* 的元素排列 。

如果未指定 *r* 或其值🄵 None，則 *r* 預設🄵 *iterable* 的長度，🄵🄵生所有可能的完整長度的排列。

輸出是*product()* 的子序列，其中重🄵元素的條目已被濾除。輸出的長度由*math.perm()* 給定，當 `0 ≤ r ≤ n` 時，長度🄵 `n! / (n - r)!`，當 `r > n` 時🄵零。

根據輸入值 *iterable* 的順序，排列的元組會按照字典順序輸出。如果輸入的 *iterable* 已排序，則輸出的元組也將按排序的順序🄵生。

元素是根據它們的位置（而非值）來🄵定其唯一性。如果輸入的元素都是獨特的，則排列中將不會有重🄵的值。

大致等價於：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) → AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) → 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.**product**(**iterables*, *repeat=1*)

輸入可🄵代物的 笛卡爾乘積

大致等價於🄵生器運算式中的巢狀 for 🄵圈。例如，`product(A, B)` 的回傳結果與 `((x,y) for x in A for y in B)` 相同。

巢狀🄵圈的循環類似於里程表，最右邊的元素在每次🄵代時前進。這種模式會建立字典順序，因此如果輸入的 iterables 已排序，則輸出的乘積元組也將按排序的順序🄵生。

要計算可🄵代物件自身的乘積，可以使用可選的 *repeat* 關鍵字引數來指定重🄵次數。例如，`product(A, repeat=4)` 與 `product(A, A, A, A)` 相同。

此函式大致等價於以下的程式碼，不同之處在於真正的實作不會在記憶體中建立中間結果：

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') → Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) → 000 001 010 011 100 101 110 111

    if repeat < 0:
        raise ValueError('repeat argument cannot be negative')
    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

在*product()* 執行之前，它會完全消耗輸入的 iterables，🄵將值的池 (pools of values) 保存在記憶體中以🄵生乘積。因此，它僅對有限的輸入有用。

itertools.**repeat**(*object*[, *times*])

建立一個🄵代器，反覆回傳 *object*。除非指定了 *times* 引數，否則會執行無限次。

大致等價於：

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

repeat 的常見用途是🄴 *map* 或 *zip* 提供定值的串流：

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

建立一個🄴代器，使用從 *iterable* 取得的引數計算 *function* 。當引數參數已經被「預先壓縮 (pre-zipped)」成元組時，使用此方法代替*map()*。

map() 和*starmap()* 之間的區🄵類似於 `function(a,b)` 和 `function(*c)` 之間的區🄵。大致等價於：

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

建立一個🄴代器，只在 *predicate* 🄵 true 時回傳 *iterable* 中的元素。大致等價於：

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

注意，第一個不符合條件判斷的元素將從輸入🄴代器中被消耗，且無法再存取它。如果應用程式希望在 *takewhile* 耗盡後進一步消耗輸入🄴代器，這可能會是個問題。🄵了解🄴這個問題，可以考慮使用 more-itertools 中的 before_and_after() 作🄴替代。

itertools.**tee**(*iterable*, *n=2*)

從一個 iterable 中回傳 *n* 個獨立的🄴代器。

大致等價於：

```
def tee(iterable, n=2):
    if n < 0:
        raise ValueError
    if n == 0:
        return ()
    iterator = _tee(iterable)
    result = [iterator]
    for _ in range(n - 1):
        result.append(_tee(iterator))
    return tuple(result)

class _tee:

    def __init__(self, iterable):
        it = iter(iterable)
        if isinstance(it, _tee):
```

(繼續下一頁)

(繼續上一頁)

```
            self.iterator = it.iterator
            self.link = it.link
        else:
            self.iterator = it
            self.link = [None, None]

    def __iter__(self):
        return self

    def __next__(self):
        link = self.link
        if link[1] is None:
            link[0] = next(self.iterator)
            link[1] = [None, None]
        value, self.link = link
        return value
```

當輸入的 *iterable* 已經是一個 tee 🄵代物件時，回傳的 tuple（元組）中所有成員都會被建立，就如同它們是由上游的`tee()` 呼叫所🄵生的一樣。這個「展平步驟 (flattening step)」讓巢狀的`tee()` 呼叫能共享相同的底層資料鏈，🄵以單一的更新步驟取代一連串的呼叫。

展平特性讓 tee 🄵代器具備高效的預覽能力：

```
def lookahead(tee_iterator):
    "回傳下一個值，但不推進輸入"
    [forked_iterator] = tee(tee_iterator, 1)
    return next(forked_iterator)
```

```
>>> iterator = iter('abcdef')
>>> [iterator] = tee(iterator, 1)    # 使得輸入可預覽
>>> next(iterator)                   # 往前移動🄵代器
'a'
>>> lookahead(iterator)              # 檢查下一個值
'b'
>>> next(iterator)                   # 繼續往前移動
'b'
```

`tee` 🄵代器不是執行緒安全 (threadsafe) 的。當同時使用由同一個`tee()` 呼叫所回傳的🄵代器時，即使原始的 *iterable* 是執行緒安全的，也可能引發`RuntimeError`。

此 itertool 可能需要大量的輔助儲存空間（取🄵於需要儲存多少臨時資料）。通常如果一個🄵代器在另一個🄵代器開始之前使用了大部分或全部的資料，使用`list()` 會比`tee()` 更快。

`itertools.`**`zip_longest`**`(`*iterables*, *fillvalue=None*`)`

建立一個🄵代器，聚合來自每個 *iterables* 中的元素。

如果 iterables 的長度不一，則使用 *fillvalue* 填充缺少的值。如果未指定，*fillvalue* 會預設🄵 `None`。

🄵代將持續直到最長的可🄵代物件耗盡🄵止。

大致等價於：

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
```

(繼續下一頁)

(繼續上一頁)

```
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

如果其中一個 iterables 可能是無限的，那🄵應該用🄵的可以限制呼叫次數的方法來包裝*zip_longest()* 函式（例如*islice()* 或*takewhile()*）。

10.1.2 Itertools 應用技巧

此段落展示了使用現有的 itertools 作🄵構建塊來建立擴展工具集的應用技巧。

itertools 應用技巧的主要目的是教學。這些應用技巧展示了對單個工具進行思考的各種方式——例如，`chain.from_iterable` 與攤平 (flattening) 的概念相關。這些應用技巧還提供了組合使用工具的想法——例如，`starmap()` 和 `repeat()` 如何一起工作。另外還展示了將 itertools 與*operator* 和*collections* 模組一同使用以及與🄵建 itertools（如 `map()`、`filter()`、`reversed()` 和 `enumerate()`）一同使用的模式。

應用技巧的次要目的是作🄵 itertools 的孵化器。`accumulate()`, `compress()` 和 `pairwise()` itertools 最初都是作🄵應用技巧出現的。目前，`sliding_window()`、`iter_index()` 和 `sieve()` 的應用技巧正在被測試，以確定它們是否有價值被收🄵到🄵建的 itertools 中。

幾乎所有這些應用技巧以及許多其他應用技巧都可以從 Python Package Index 上的 more-itertools 專案中安裝：

```
python -m pip install more-itertools
```

許多應用技巧提供了與底層工具集相同的高性能。透過一次處理一個元素而不是將整個可🄵代物件一次性引入記憶體，能保持優🄵的記憶體性能。以函式風格 (functional style) 將工具連接在一起，能將程式碼的數量維持在較少的情🄵。透過優先使用「向量化 (vectorized)」的構建塊而不是使用會造成直譯器負擔的 for 🄵圈和🄵生器，則能保持高速度。

```
from collections import Counter, deque
from contextlib import suppress
from functools import reduce
from math import comb, prod, sumprod, isqrt
from operator import itemgetter, getitem, mul, neg

def take(n, iterable):
    "回傳可🄵代物件的前 n 個元素🄵串列。"
    return list(islice(iterable, n))

def prepend(value, iterable):
    "在可🄵代物件開頭插入單一值。"
    # prepend(1, [2, 3, 4]) → 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "回傳 function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(function, times=None, *args):
    "重🄵呼叫一個帶指定引數的函式。"
    if times is None:
        return starmap(function, repeat(args))
```

(繼續下一頁)

(繼續上一頁)

```
    return starmap(function, repeat(args, times))

def flatten(list_of_lists):
    "將巢狀結構攤平一層。"
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "回傳序列的元素重🄵 n 次。"
    return chain.from_iterable(repeat(tuple(iterable), n))

def loops(n):
    "執行 n 次的🄵圈。類似 range(n) 但不建立整數序列。"
    # for _ in loops(100): ...
    return repeat(None, n)

def tail(n, iterable):
    "回傳一個🄵代器，🄵代最後 n 個元素。"
    # tail(3, 'ABCDEFG') → E F G
    return iter(deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "將🄵代器往前推進 n 步。如果 n 🄵 None，則完全消耗。"
    # 使用以 C 語言的速度消耗🄵代器的函式。
    if n is None:
        deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "回傳第 n 個元素或預設值。"
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "給定一個回傳 True 或 False 的判斷函式，計算🄵 True 的結果。"
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "回傳第一個🄵 true 的值，若無則回傳*預設值*。"
    # first_true([a,b,c], x) → a or b or c or x
    # first_true([a,b], x, f) → a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "回傳 True，如果所有元素兩兩相等。"
    # all_equal('4🄵🄵🄵🄵', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
    "🄵生唯一的元素，🄵保留原始順序。只記住剛看見的元素。"
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(itemgetter(0), groupby(iterable))
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "🄵生唯一的元素，🄵保留原始順序。記住所有曾見過的元素。"
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    if key is None:
```

(繼續下一頁)

(繼續上一頁)

```
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen.add(k)
                yield element

def unique(iterable, key=None, reverse=False):
    "🄵生排序後的不重🄵元素。支援不可雜🄵的輸入。"
    # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
    sequenced = sorted(iterable, key=key, reverse=reverse)
    return unique_justseen(sequenced, key=key)

def sliding_window(iterable, n):
    "將資料收集成重🄵的固定長度區段或區塊。"
    # sliding_window('ABCDEFG', 4) → ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "將資料收集成不重🄵的固定長度區段或區塊。"
    # grouper('ABCDEFG', 3, fillvalue='x') → ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') → ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') → ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "以循環方式依序輸入可🄵代物件，直到全部耗盡。"
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # 演算法出自 George Sakkis
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def subslices(seq):
    "回傳序列的所有連續非空子切片。"
    # subslices('ABCD') → A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(getitem, repeat(seq), slices)

def iter_index(iterable, value, start=0, stop=None):
    "回傳在序列或可🄵代物件中某值出現的索引位置。"
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
```

(繼續下一頁)

(繼續上一頁)

```
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(function, exception, first=None):
    "將一個 call-until-exception 轉🄵🄵🄵代器介面。"
    # iter_except(d.popitem, KeyError) → 非阻塞的字典🄵代器
    with suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield function()
```

以下的應用技巧具有更多的數學風格:

```
def multinomial(*counts):
    "多重集合的不同排列數。"
    # Counter('abracadabra').values() → 5 2 2 1 1
    # multinomial(5, 2, 2, 1, 1) → 83160
    return prod(map(comb, accumulate(counts), counts))

def powerset(iterable):
    "來自可🄵代物件的子序列，從最短到最長。"
    # powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "將輸入值的平方加總。"
    # sum_of_squares([10, 20, 30]) → 1400
    return sumprod(*tee(iterable))

def reshape(matrix, columns):
    "將 2 維矩陣重新塑形🄵指定的行數。"
    # reshape([(0, 1), (2, 3), (4, 5)], 3) →  (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), columns, strict=True)

def transpose(matrix):
    "交🄵 2 維矩陣的列和行。"
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "矩陣相乘。"
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """兩個可🄵代物件的離散🄵性🄵積。
    等同於多項式相乘。

    在數學上🄵積是可交🄵的；但輸入的處理方式不同。
```

(繼續下一頁)

(繼續上一頁)

```
    訊號以惰性方式被讀取，且可以是無限；核心會在計算開始前被全部讀取。

    文章：https://betterexplained.com/articles/intuitive-convolution/
    影片：https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → 移動平均（模糊）
    # convolve(data, [1/2, 0, -1/2]) → 一階導數估計
    # convolve(data, [1, -2, 1]) → 二階導數估計
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """由多項式的根計算其🄵數。

       (x - 5) (x + 4) (x - 3)  展開🄵:  x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(neg, roots))
    return list(reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """在指定值計算多項式的值。

    此方法在數值穩定性上比 Horner 方法更好。
    """
    # 計算 x³ -4x² -17x + 60 在 x = 5 時的值
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
    powers = map(pow, repeat(x), reversed(range(n)))
    return sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """計算多項式的一階導數。

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(mul, coefficients, powers))

def sieve(n):
    "小於 n 的質數。"
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "n 的質因數。"
    # factor(99) → 3 3 11
```

(繼續下一頁)

(繼續上一頁)

```
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def is_prime(n):
    "回傳 True, 若 n 🄵質數。"
    # is_prime(1_000_000_000_000_403) → True
    return n > 1 and next(factor(n)) == n

def totient(n):
    "計算不大於 n 且與 n 互質的自然數個數。"
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 因爲 len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```

10.2 `functools` --- 可呼叫物件上的高階函式與操作

原始碼： Lib/functools.py

---

functools 模組用於高階函式：作用於或回傳其他函式的函式。一般來🄵，任何可呼叫物件都可以被視🄵用於此模組的函式。

functools 模組定義了以下函式：

@functools.**cache**(*user_function*)

簡單的輕量級無🄵結函式快取 (Simple lightweight unbounded function cache)。有時稱之🄵 "memoize"（記憶化）。

和 `lru_cache(maxsize=None)` 回傳相同的值，🄵函式引數建立一個字典查找的薄包裝器。因🄵它永遠不需要🄵🄵舊值，所以這比有大小限制的 *lru_cache()* 更小、更快。

舉例來🄵：

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # 🄵有先前的快取結果，會進行 11 次遞🄵呼叫
3628800
>>> factorial(5)       # 只查詢快取的結果
120
>>> factorial(12)      # 進行兩次新的遞🄵呼叫，其他 10 次是快取的
479001600
```

該快取是執行緒安全的 (threadsafe)，因此包裝的函式可以在多個執行緒中使用。這意味著底層資料結構在🄵行更新期間將保持連貫 (coherent)。

如果另一個執行緒在初始呼叫完成🄵快取之前進行額外的呼叫，則包裝的函式可能會被多次呼叫。

在 3.9 版被加入.

@functools.**cached_property**(*func*)

將類🄵的一個方法轉🄵🄵屬性 (property)，其值會計算一次，然後在實例的生命🄵期🄵快取🄵普通屬性。類似*property()*，但增加了快取機制。對於除使用該裝飾器的屬性外實質上幾乎是不可變 (immutable) 的實例，針對其所需要繁重計算會很有用。

範例：

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

cached_property() 的機制與*property()* 有所不同。除非定義了 setter，否則常規屬性會阻止屬性的寫入。相反地，*cached_property* 則允許寫入。

cached_property 裝飾器僅在查找時且僅在同名屬性不存在時運行。當它運行時，*cached_property* 會寫入同名的屬性。後續屬性讀取和寫入優先於 *cached_property* 方法，🄵且它的工作方式與普通屬性類似。

可以透過🄵除屬性來清除快取的值，這使得 *cached_property* 方法可以再次運行。

cached_property 無法防止多執行緒使用中可能出現的競🄵條件 (race condition)。getter 函式可以在同一個實例上運行多次，最後一次運行會設定快取的值。所以快取的屬性最好是🄵等的 (idempotent)，或者在一個實例上運行多次不會有害，就不會有問題。如果同步是必要的，請在裝飾的 getter 函式🄵部或在快取的屬性存取周圍實作必要的鎖。

請注意，此裝飾器會干擾 **PEP 412** 金鑰共用字典的操作。這意味著實例字典可能比平常🄵用更多的空間。

此外，此裝飾器要求每個實例上的 `__dict__` 屬性是可變對映 (mutable mapping)。這意味著它不適用於某些型🄵，例如元類🄵 (metaclass)（因🄵型🄵實例上的 `__dict__` 屬性是類🄵命名空間的唯讀代理），以及那些指定 `__slots__` 而不包含 `__dict__` 的型🄵作🄵有定義的插槽之一（因🄵此種類🄵根本不提供 `__dict__` 屬性）。

如果可變對映不可用或需要金鑰共享以節省空間，則也可以透過在*lru_cache()* 之上堆🄵*property()* 來實作類似於*cached_property()* 的效果。請參🄵 faq-cache-method-calls 以了解有關這與*cached_property()* 間不同之處的更多詳細資訊。

在 3.8 版被加入.

在 3.12 版的變更: 在 Python 3.12 之前，`cached_property` 包含一個未以文件記🄵的鎖，以確保在多執行緒使用中能保證 getter 函式對於每個實例只會執行一次。然而，鎖是針對每個屬性，而不是針對每個實例，這可能會導致無法被接受的嚴重鎖🄵用 (lock contention)。在 Python 3.12+ 中，此鎖已被🄵除。

functools.**cmp_to_key**(*func*)

將舊式比較函式轉🄵🄵*鍵函式*，能與接受鍵函式的工具一起使用（例如*sorted()*、*min()*、*max()*、*heapq.nlargest()*、*heapq.nsmallest()*、*itertools.groupby()*）。此函式主要作🄵轉🄵工具，用於從有支援使用比較函式的 Python 2 轉🄵成的程式。

比較函式是任何能接受兩個引數、對它們進行比較，🄵回傳負數（小於）、零（相等）或正數（大於）的可呼叫物件。鍵函式是接受一個引數🄵回傳另一個用作排序鍵之值的可呼叫物件。

範例：

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

有關排序範例和簡短的排序教學，請參🄵 sortinghowto。

在 3.2 版被加入.

@functools.**lru_cache**(*user_function*)

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

以記憶化可呼叫物件來包裝函式的裝飾器，最多可省去 *maxsize* 個最近的呼叫。當使用相同引數定期呼叫繁重的或 I/O 密集的函式時，它可以節省時間。

該快取是執行緒安全的 (threadsafe)，因此包裝的函式可以在多個執行緒中使用。這意味著底層資料結構在🄵行更新期間將保持連貫 (coherent)。

如果另一個執行緒在初始呼叫完成🄵快取之前進行額外的呼叫，則包裝的函式可能會被多次呼叫。

由於字典用於快取結果，因此函式的位置引數和關鍵字引數必須是可雜🄵的。

不同的引數模式可以被認🄵是具有不同快取條目的不同呼叫。例如，`f(a=1, b=2)` 和 `f(b=2, a=1)` 的關鍵字引數順序不同，🄵且可能有兩個不同的快取條目。

如果指定了 *user_function*，則它必須是個可呼叫物件。這使得 *lru_cache* 裝飾器能🄵直接應用於使用者函式，將 *maxsize* 保留🄵其預設值 128：

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

如果 *maxsize* 設定🄵 `None`，則 LRU 功能將被停用，且快取可以無限制地成長。

如果 *typed* 設定🄵 true，不同型🄵的函式引數將會被單獨快取起來。如果 *typed* 🄵 false，則實作通常會將它們視🄵等效呼叫，🄵且僅快取單一結果。（某些型🄵，例如 *str* 和 *int* 可能會被單獨快取起來，即使 *typed* 🄵 false。）

請注意，型🄵特🄵性 (type specificity) 僅適用於函式的直接引數而不是其🄵容。純量 (scalar) 引數 `Decimal(42)` 和 `Fraction(42)` 被視🄵具有不同結果的不同呼叫。相反地，元組引數 `('answer', Decimal(42))` 和 `('answer', Fraction(42))` 被視🄵等效。

包裝的函式使用一個 `cache_parameters()` 函式來進行偵測，該函式回傳一個新的*dict* 以顯示 *maxsize* 和 *typed* 的值。這僅能顯示資訊，改變其值不會有任何效果。🄵了輔助測量快取的有效性🄵調整 *maxsize* 參數，包裝的函式使用了一個 `cache_info()` 函式來做檢測，該函式會回傳一個附名元組來顯示 *hits*、*misses*、*maxsize* 和 *currsize*。裝飾器還提供了一個 `cache_clear()` 函式來清除或使快取失效。

原本的底層函式可以透過 `__wrapped__` 屬性存取。這對於要自我檢查 (introspection)、繞過快取或使用不同的快取重新包裝函式時非常有用。

快取會保留對引數和回傳值的參照，直到快取過時 (age out) 或快取被清除🄵止。

如果方法被快取起來，則 `self` 實例引數將包含在快取中。請參🄵 faq-cache-method-calls

當最近的呼叫是即將發生之呼叫的最佳預測因子時（例如新聞伺服器上最受歡迎的文章往往每天都會發生變化），LRU (least recently used) 快取能發揮最好的效果。快取的大小限制可確保快取不會在長時間運行的行程（例如 Web 伺服器）上無限制地成長。

一般來🄵，僅當你想要重🄵使用先前計算的值時才應使用 LRU 快取。因此，快取具有 side-effects 的函式、需要在每次呼叫時建立不同可變物件的函式（例如🄵生器和非同步函式）或不純函式（impure function，例如 time() 或 random()）是🄵有意義的。

🄵態網頁🄵容的 LRU 快取範例：

```
@lru_cache(maxsize=32)
def get_pep(num):
    '取得 Python 改進提案的文字'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
```

(繼續下一頁)

(繼續上一頁)

```
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

使用快取來實作動態規劃 (dynamic programming) 技法以有效率地計算費波那契數 (Fibonacci numbers) 的範例：

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

在 3.2 版被加入.

在 3.3 版的變更: 新增 *typed* 選項。

在 3.8 版的變更: 新增 *user_function* 選項。

在 3.9 版的變更: 新增 `cache_parameters()` 函式。

`@functools.total_ordering`

給定一個定義一個或多個 rich comparison 排序方法的類🄵，該類🄵裝飾器會提供其余部分。這簡化了指定所有可能的 rich comparison 操作所涉及的工作：

類🄵必須定義 `__lt__()`、`__le__()`、`__gt__()` 或 `__ge__()` 其中之一。此外，該類🄵應該提供 `__eq__()` 方法。

舉例來🄵：

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

備🄵

雖然此裝飾器可以輕松建立能好好運作的完全有序型🄵 (totally ordered types)，但它的確以衍生比較方法的執行速度較慢和堆🄵追🄵 (stack trace) 較🄵雜做🄵其代價。如果效能基准測試顯示這是給定應用程式的效能瓶頸，那🄵實作全部六種 rich comparison 方法通常能輕松地提升速度。

> **備🄵**
>
> 此裝飾器不會嘗試覆寫類🄴或其*超類🄴 (superclass)* 中宣告的方法。這意味著如果超類🄴定義了比較運算子，*total_ordering* 將不會再次實作它，即使原始方法是抽象的。

在 3.2 版被加入.

在 3.4 版的變更: 現在支援從底層對於未識🄴型🄴的比較函式回傳 `NotImplemented`。

`functools.`**`Placeholder`**

A singleton object used as a sentinel to reserve a place for positional arguments when calling *`partial()`* and *`partialmethod()`*.

在 3.14 版被加入.

`functools.`**`partial`**(*func*, */*, **args*, ***keywords*)

回傳一個新的*partial 物件*，它在被呼叫時的行🄴類似於使用位置引數 *args* 和關鍵字引數 *keywords* 呼叫的 *func*。如果向呼叫提供更多引數，它們將被附加到 *args*。如果提供了額外的關鍵字引數，它們會擴充🄴覆寫 *keywords*。大致相當於：

```
def partial(func, /, *args, **keywords):
    def newfunc(*more_args, **more_keywords):
        return func(*args, *more_args, **(keywords | more_keywords))
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

`partial()` 用於部分函式應用程序，它「凍結」函式引數和/或關鍵字的某些部分，從而🄴生具有簡化簽名的新物件。例如，*`partial()`* 可用來建立可呼叫函式，其行🄴類似於*`int()`* 函式，其中 *base* 引數預設🄴 2：

```
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

If *`Placeholder`* sentinels are present in *args*, they will be filled first when `partial()` is called. This makes it possible to pre-fill any positional argument with a call to `partial()`; without `Placeholder`, only the chosen number of leading positional arguments can be pre-filled.

If any `Placeholder` sentinels are present, all must be filled at call time:

```
>>> say_to_world = partial(print, Placeholder, Placeholder, "world!")
>>> say_to_world('Hello', 'dear')
Hello dear world!
```

Calling `say_to_world('Hello')` raises a *`TypeError`*, because only one positional argument is provided, but there are two placeholders that must be filled in.

If `partial()` is applied to an existing `partial()` object, `Placeholder` sentinels of the input object are filled in with new positional arguments. A placeholder can be retained by inserting a new `Placeholder` sentinel to the place held by a previous `Placeholder`:

```
>>> from functools import partial, Placeholder as _
>>> remove = partial(str.replace, _, _, '')
>>> message = 'Hello, dear dear world!'
>>> remove(message, ' dear')
'Hello, world!'
>>> remove_dear = partial(remove, _, ' dear')
>>> remove_dear(message)
```

(繼續下一頁)

(繼續上一頁)

```
'Hello, world!'
>>> remove_first_dear = partial(remove_dear, _, 1)
>>> remove_first_dear(message)
'Hello, dear world!'
```

`Placeholder` cannot be passed to `partial()` as a keyword argument.

在 3.14 版的變更: Added support for *`Placeholder`* in positional arguments.

class functools.**partialmethod**(*func*, */*, **args*, ***keywords*)

回傳一個新的*`partialmethod`* 描述器 (descriptor)，其行🄵類似於*`partial`*，只不過它被設計🄵用於方法定義而不能直接呼叫。

func 必須是一個*descriptor* 或可呼叫物件（兩者兼具的物件，就像普通函式一樣，會被當作描述器處理）。

當 *func* 是描述器時（例如普通的 Python 函式、*`classmethod()`*、*`staticmethod()`*、*`abstractmethod()`* 或*`partialmethod`* 的另一個實例），對 `__get__` 的呼叫將被委🄵 (delegated) 給底層描述器，且一個適當的*partial 物件*會被作🄵結果回傳。

當 *func* 是非描述器可呼叫物件 (non-descriptor callable) 時，會動態建立適當的🄵結方法 (bound method)。當被作🄵方法使用時，其行🄵類似於普通的 Python 函式：*self* 引數將作🄵第一個位置引數插入，甚至會在提供給*`partialmethod`* 建構函式的 *args* 和 *keywords* 的前面。

範例：

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

在 3.4 版被加入.

functools.**reduce**(*function*, *iterable*, */*[, *initial*])

從左到右，將兩個引數的 *function* 累加運用到 *iterable* 的項目上，從而將可🄵代物件🄵少🄵單一值。例如，`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` 會計算出 `((((1+2)+3)+4)+5)`。左邊的引數 *x* 是累積值，右邊的引數 *y* 是來自 *iterable* 的更新值。如果可選的 *initial* 存在，則在計算中會將其放置在可🄵代物件的項目之前，🄵在可🄵代物件🄵空時作🄵預設值。如果未給定 *initial* 且 *iterable* 僅包含一個項目，則回傳第一個項目。

大致相當於：

```
initial_missing = object()

def reduce(function, iterable, /, initial=initial_missing):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
```

(繼續下一頁)

(繼續上一頁)

```
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

請參🄵*itertools.accumulate()* 以了解🄵生 (yield) 所有中間值 (intermediate value) 的🄵代器。

在 3.14 版的變更: *initial* is now supported as a keyword argument.

`@functools.`**singledispatch**

將函式轉🄵🄵單一調度泛型函式。

若要定義泛型函式，請使用 `@singledispatch` 裝飾器對其裝飾。請注意，使用 `@singledispatch` 定義函式時，分派調度 (dispatch) 是發生在第一個引數的型🄴上：

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

若要🄵函式新增過載實作，請使用泛型函式的 `register()` 屬性，該屬性可用作裝飾器。對於以型🄵來🄵釋的函式，裝飾器將自動推斷第一個引數的型🄵：

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

也可以使用*typing.Union*：

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

對於不使用型🄵🄵釋的程式碼，可以將適當的型🄵引數明確傳遞給裝飾器本身：

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
```

(繼續下一頁)

(繼續上一頁)

```
...     print(arg.real, arg.imag)
...
```

For code that dispatches on a collections type (e.g., `list`), but wants to typehint the items of the collection (e.g., `list[int]`), the dispatch type should be passed explicitly to the decorator itself with the typehint going into the function definition:

```
>>> @fun.register(list)
... def _(arg: list[int], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

備🄵

At runtime the function will dispatch on an instance of a list regardless of the type contained within the list i.e. `[1,2,3]` will be dispatched the same as `["foo", "bar", "baz"]`. The annotation provided in this example is for static type checkers only and has no runtime impact.

若要🄵用🄵🄵*lambdas* 和預先存在的函式，*`register()`* 屬性也能以函式形式使用：

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

`register()` 屬性回傳未加裝飾器的函式。這讓使得裝飾器堆🄵 (decorator stacking)、*`pickling`* 以及🄵每個變體獨立建立單元測試成🄵可能：

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

呼叫時，泛型函式會分派第一個引數的型🄵：

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

如果🄵有🄵特定型🄵🄵🄵實作，則使用其方法解析順序 (method resolution order) 來尋找更通用的實作。用 `@singledispatch` 裝飾的原始函式是🄵基底*object* 型🄵🄵🄵的，這意味著如果🄵有找到更好的實作就會使用它。

如果一個實作有被🄵🄵到一個抽象基底類🄵，則基底類🄵的🄵擬子類🄵將被分派到該實作：

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

若要檢查泛型函式將🄵給定型🄵選擇哪種實作，請使用 `dispatch()` 屬性：

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

若要存取所有已🄵🄵的實作，請使用唯讀 `registry` 屬性：

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

在 3.4 版被加入.

在 3.7 版的變更: *register()* 屬性現在支援使用型🄵🄵釋。

在 3.11 版的變更: *register()* 屬性現在支援使用*typing.Union* 型🄵🄵釋。

class functools.**singledispatchmethod**(*func*)

將方法轉🄵🄵單一調度泛型函式。

若要定義泛型方法，請使用 `@singledispatchmethod` 裝飾器對其裝飾。請注意，使用 `@singledispatchmethod` 定義函式時，分派調度是發生在第一個非 *self* 或非 *cls* 引數的型🄵上：

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` 支援與其他裝飾器巢狀使用 (nesting)，例如*@classmethod*。請注意，🄵了使 `dispatcher.register` 可用，`singledispatchmethod` 必須是最外面的裝飾器。以下範例是 `Negator` 類🄵，其 `neg` 方法🄵結到該類🄵，而不是該類🄵的實例：

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

相同的模式可用於其他類似的裝飾器：*@staticmethod*、*@abstractmethod* 等。

在 3.8 版被加入.

functools.**update_wrapper**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

更新 *wrapper* 函式，使其看起來像 *wrapped* 函式。可選引數是元組，用於指定原始函式的哪些屬性直接賦值給包裝函式上的匹配屬性，以及包裝函式的哪些屬性使用原始函式中的對應屬性進行更新。這些引數的預設值是模組層級的常數 `WRAPPER_ASSIGNMENTS`（它賦值給包裝函式的 `__module__`、`__name__`、`__qualname__`、`__annotations__`、`__type_params__` 和 `__doc__` 文件字串 (docstring)）和 `WRAPPER_UPDATES`（更新包裝器函式的 `__dict__`，即實例字典）。

🄵了允許出於🄵省 (introspection) 和其他目的所對原始函式的存取（例如繞過快取裝飾器，如*lru_cache()*），此函式會自動向包裝器新增 `__wrapped__` 屬性，該包裝器參照被包裝的函式。

此函式的主要用途是在*decorator* 函式中，它包裝函式🄵回傳包裝器。如果包裝器函式未更新，則回傳函式的元資料 (metadata) 將反映包裝器定義而非原始函式定義，這通常不太會有幫助。

update_wrapper() 可以與函式以外的可呼叫物件一起使用。被包裝的物件中缺少的 *assigned* 或 *updated* 中指定的任何屬性都將被忽略（即此函式不會嘗試在包裝器函式上設定它們）。如果包裝函式本身缺少 *updated* 中指定的任何屬性，仍然會引發*AttributeError*。

在 3.2 版的變更: 現在會自動新增 `__wrapped__` 屬性。現在預設會🄵🄵 `__annotations__` 屬性。缺少的屬性不再觸發*AttributeError*。

在 3.4 版的變更: `__wrapped__` 屬性現在都會參照包裝函式，即便函式有定義 `__wrapped__` 屬性。(參見 bpo-17482)

在 3.12 版的變更: 現在預設會🄵🄵 `__type_params__` 屬性。

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

這是一個方便的函式，用於在定義包裝器函式時呼叫*update_wrapper()* 作🄵函式裝飾器。它相當於 `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`。例如：

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
```

(繼續下一頁)

(繼續上一頁)

```
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果不使用這個裝飾器工廠 (decorator factory)，範例函式的名稱將會是 `'wrapper'`，🄵且原始 `example()` 的文件字串將會遺失。

10.2.1 `partial` 物件

partial 物件是由*partial()* 所建立的可呼叫物件。它們有三個唯讀屬性：

partial.**func**

一個可呼叫的物件或函式。對*partial* 物件的呼叫將被轉送到帶有新引數和關鍵字的*func*。

partial.**args**

最左邊的位置引數將會被加入到提供給*partial* 物件呼叫的位置引數的前面。

partial.**keywords**

呼叫*partial* 物件時將提供的關鍵字引數。

partial 物件與函式物件類似，因🄵它們是可呼叫的、可弱參照的 (weak referencable) 且可以具有屬性的。但有一些重要的區🄵，例如，*__name__* 和*__doc__* 屬性不會自動建立。

10.3 `operator` --- 標準運算子替代函式

原始碼： Lib/operator.py

---

operator module（模組）提供了一套與 Python 原生運算子對應的高效率函式。例如，`operator.add(x, y)` 與表示式 `x+y` 相同。許多函式名與特殊方法名相同，只是🄵有雙底🄵。🄵了向後相容，許多包含雙底🄵的函式被保留了下來，但🄵了易於表達，建議使用🄵有雙底🄵的函式。

函式種類有物件的比較運算、邏輯運算、數學運算以及序列運算。

物件比較函式適用於所有物件，函式根據它們對應的 rich comparison 運算子命名：

operator.**lt**(*a*, *b*)

operator.**le**(*a*, *b*)

operator.**eq**(*a*, *b*)

operator.**ne**(*a*, *b*)

operator.**ge**(*a*, *b*)

operator.**gt**(*a*, *b*)

operator.**__lt__**(*a*, *b*)

operator.**__le__**(*a*, *b*)

operator.**__eq__**(*a*, *b*)

operator.**__ne__**(*a*, *b*)

operator.**__ge__**(*a*, *b*)

operator.**__gt__**(*a*, *b*)

在 *a* 和 *b* 之間進行"rich comparison"。具體來🄴，`lt(a, b)` 與 `a < b` 相同，`le(a, b)` 與 `a <= b` 相同，`eq(a, b)` 與 `a == b` 相同，`ne(a, b)` 與 `a != b` 相同，`gt(a, b)` 與 `a > b` 相同，`ge(a, b)` 與 `a >= b` 相同。注意這些函式可以回傳任何值，無論它是否可當作 boolean（布林）值。關於 rich comparison 的更多資訊請參考 comparisons。

邏輯運算通常也適用於所有物件，🄴且支援真值檢測、識🄴性測試和 boolean 運算：

operator.**not_**(*obj*)

operator.**__not__**(*obj*)

回傳 `not` *obj* 的結果。（請注意物件實例🄴🄴有 `__not__()` method（方法）；只有直譯器核心定義此操作。結果會受 `__bool__()` 和 `__len__()` method 影響。）

operator.**truth**(*obj*)

如果 *obj* 🄴真值則回傳 *True*，否則回傳 *False*。這等價於使用 *bool* 建構函式。

operator.**is_**(*a*, *b*)

回傳 `a is b`。測試物件識🄴性。

operator.**is_not**(*a*, *b*)

回傳 `a is not b`。測試物件識🄴性。

operator.**is_none**(*a*)

回傳 `a is None`。測試物件識🄴性。

在 3.14 版被加入.

operator.**is_not_none**(*a*)

回傳 `a is not None`。測試物件識🄴性。

在 3.14 版被加入.

數學和位元運算的種類是最多的：

operator.**abs**(*obj*)

operator.**__abs__**(*obj*)

回傳 *obj* 的🄴對值。

operator.**add**(*a*, *b*)

operator.**__add__**(*a*, *b*)

對於數字 *a* 和 *b*，回傳 `a + b`。

operator.**and_**(*a*, *b*)

operator.**__and__**(*a*, *b*)

回傳 *x* 和 *y* 位元運算與 (and) 的結果。

operator.**floordiv**(*a*, *b*)

operator.**__floordiv__**(*a*, *b*)

回傳 `a // b`。

operator.**index**(*a*)

operator.**__index__**(*a*)

回傳 *a* 轉🄵🄵整數的結果。等價於 `a.__index__()`。

在 3.10 版的變更: 結果總是 *int* 型🄴。在過去的版本中，結果可能🄴 `int` 子類🄴的實例。

operator.**inv**(*obj*)

operator.**invert**(*obj*)

operator.**__inv__**(*obj*)

operator.**__invert__**(*obj*)
回傳數字 *obj* 按位元取反 (inverse) 的結果。這等價於 `~obj`。

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
回傳 *a* 左移 *b* 位的結果。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
回傳 `a % b`。

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
對於數字 *a* 和 *b*，回傳 `a * b`。

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
回傳 `a @ b`。

在 3.5 版被加入.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
回傳 *obj* 取負值的結果 (`-obj`)。

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
回傳 *a* 和 *b* 按位元或 (or) 的結果。

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
回傳 *obj* 取正的結果 (`+obj`)。

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
對於數字 *a* 和 *b*，回傳 `a ** b`。

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
回傳 *a* 右移 *b* 位的結果。

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
回傳 `a - b`。

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
回傳 `a / b`，例如 2/3 將等於.66 而不是 0。這也被稱F「真」除法。

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
回傳 *a* 和 *b* 按位元F或 (exclusive or) 的結果。

適用於序列的操作（其中一些也適用於對映 (mapping)），包括：

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
對於序列 *a* 和 *b*，回傳 `a + b`。

operator.**contains**(*a*, *b*)

operator.**__contains__**(*a*, *b*)

回傳 `b in a` 檢測的結果。請注意運算元是反序的。

operator.**countOf**(*a*, *b*)

回傳 *b* 在 *a* 中的出現次數。

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

移除 *a* 中索引🄵 *b* 的值。

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

回傳 *a* 中索引🄵 *b* 的值。

operator.**indexOf**(*a*, *b*)

回傳 *b* 在 *a* 中首次出現所在的索引。

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

將 *a* 中索引🄵 *b* 的值設🄵 *c*。

operator.**length_hint**(*obj*, *default=0*)

回傳物件 *obj* 的估計長度。首先嘗試回傳其實際長度，再使用 `object.__length_hint__()` 得出估計值，最後才是回傳預設值。

在 3.4 版被加入.

以下操作適用於可呼叫物件：

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

回傳 `obj(*args, **kwargs)`。

在 3.11 版被加入.

operator module 還定義了一些用於常規屬性和條目查詢的工具。這些工具適合用來編寫快速欄位提取器以作🄵*map()*、*sorted()*、*itertools.groupby()* 或其他需要函式引數的函式之引數。

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

回傳一個可從運算元中取得 *attr* 的可呼叫 (callable) 物件。如果請求了一個以上的屬性，則回傳一個包含屬性的 tupple（元組）。屬性名稱還可包含點號。例如：

- 在 `f = attrgetter('name')` 之後，呼叫 `f(b)` 將回傳 `b.name`。
- 在 `f = attrgetter('name', 'date')` 之後，呼叫 `f(b)` 將回傳 `(b.name, b.date)`。
- 在 `f = attrgetter('name.first', 'name.last')` 之後，呼叫 `f(b)` 將回傳 `(b.name.first, b.name.last)`。

等價於：

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('屬性名稱必須是一個字串')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g
```

(繼續下一頁)

(繼續上一頁)

```
def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

回傳一個使用運算元的 `__getitem__()` 方法從運算元中取得 *item* 的可呼叫物件。如果指定了多個條目，則回傳一個查詢值的 tupple。例如：

- 在 `f = itemgetter(2)` 之後，呼叫 `f(r)` 將回傳 `r[2]`。
- 在 `g = itemgetter(2, 5, 3)` 之後，呼叫 `g(r)` 將回傳 `(r[2], r[5], r[3])`。

等價於：

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

傳入的條目可以🄴運算元的 `__getitem__()` 所接受的任何型🄵。dictionary（字典）接受任意可雜🄵的值。list、tupple 和字串接受索引或切片：

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

使用 *itemgetter()* 從 tuple 中提取特定欄位的例子：

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

回傳一個在運算元上呼叫 *name* method 的可呼叫物件。如果給定額外的引數和/或關鍵字引數，它們也將被傳給該 method。例如：

- 在 `f = methodcaller('name')` 之後，呼叫 `f(b)` 將回傳 `b.name()`。
- 在 `f = methodcaller('name', 'foo', bar=1)` 之後，呼叫 `f(b)` 將回傳 `b.name('foo', bar=1)`。

等價於：

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
```

(繼續下一頁)

(繼續上一頁)

```
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 運算子與函式間的對映

以下表格表示了抽象運算是如何對應於 Python 語法中的運算子和*operator* module 中的函式。

運算	語法	函式
加法	`a + b`	`add(a, b)`
字串串接	`seq1 + seq2`	`concat(seq1, seq2)`
包含性檢測	`obj in seq`	`contains(seq, obj)`
除法	`a / b`	`truediv(a, b)`
除法	`a // b`	`floordiv(a, b)`
按位元與 (And)	`a & b`	`and_(a, b)`
按位元互斥或 (Exclusive Or)	`a ^ b`	`xor(a, b)`
按位元取反 (Inversion)	`~ a`	`invert(a)`
按位元或 (Or)	`a | b`	`or_(a, b)`
取🄵	`a ** b`	`pow(a, b)`
識🄵性	`a is b`	`is_(a, b)`
識🄵性	`a is not b`	`is_not(a, b)`
識🄵性	`a is None`	`is_none(a)`
識🄵性	`a is not None`	`is_not_none(a)`
索引賦值	`obj[k] = v`	`setitem(obj, k, v)`
索引🄵除	`del obj[k]`	`delitem(obj, k)`
索引取值	`obj[k]`	`getitem(obj, k)`
左移	`a << b`	`lshift(a, b)`
模除 (Modulo)	`a % b`	`mod(a, b)`
乘法	`a * b`	`mul(a, b)`
矩陣乘法	`a @ b`	`matmul(a, b)`
反相（算術）	`- a`	`neg(a)`
反相（邏輯）	`not a`	`not_(a)`
正數	`+ a`	`pos(a)`
右移	`a >> b`	`rshift(a, b)`
切片賦值	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
切片🄵除	`del seq[i:j]`	`delitem(seq, slice(i, j))`
切片取值	`seq[i:j]`	`getitem(seq, slice(i, j))`
字串格式化	`s % obj`	`mod(s, obj)`
🄵法	`a - b`	`sub(a, b)`
真值檢測	`obj`	`truth(obj)`
比較大小	`a < b`	`lt(a, b)`
比較大小	`a <= b`	`le(a, b)`
相等性	`a == b`	`eq(a, b)`
不等性	`a != b`	`ne(a, b)`
比較大小	`a >= b`	`ge(a, b)`
比較大小	`a > b`	`gt(a, b)`

10.3.2 原地 (in-place) 運算子

許多運算都有「原地」版本。以下列出的是提供對原地運算子（與一般語法相比）更底層存取的函式，例如*statement* `x += y` 相當於 `x = operator.iadd(x, y)`。🄵一種方式來講就是 `z = operator.iadd(x, y)` 等價於🄵合陳述式 `z = x; z += y`。

在這些例子中，請注意當呼叫一個原地方法時，運算和賦值是分成兩個步驟來執行的。下面列出的原地函式只執行第一步，即呼叫原地方法，第二步賦值則不加處理。

對於不可變 (immutable) 的目標例如字串、數字和 tupple，更新的值會被計算，但不會被再被賦值給輸入變數：

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

對於可變 (mutable) 的目標例如 list 和 dictionary，原地方法將執行更新，因此不需要後續賦值操作：

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

a = iadd(a, b) 等價於 a += b。

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

a = iand(a, b) 等價於 a &= b。

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

a = iconcat(a, b) 等價於 a += b，其中 *a* 和 *b* 🄵序列。

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

a = ifloordiv(a, b) 等價於 a //= b。

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

a = ilshift(a, b) 等價於 a <<= b。

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

a = imod(a, b) 等價於 a %= b。

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

a = imul(a, b) 等價於 a *= b。

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

a = imatmul(a, b) 等價於 a @= b。

在 3.5 版被加入.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

a = ior(a, b) 等價於 a |= b。

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

a = ipow(a, b) 等價於 a **= b。

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

a = irshift(a, b) 等價於 a >>= b。

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

a = isub(a, b) 等價於 a -= b。

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

a = itruediv(a, b) 等價於 a /= b。

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

a = ixor(a, b) 等價於 a ^= b。

CHAPTER 11

檔案與目F存取

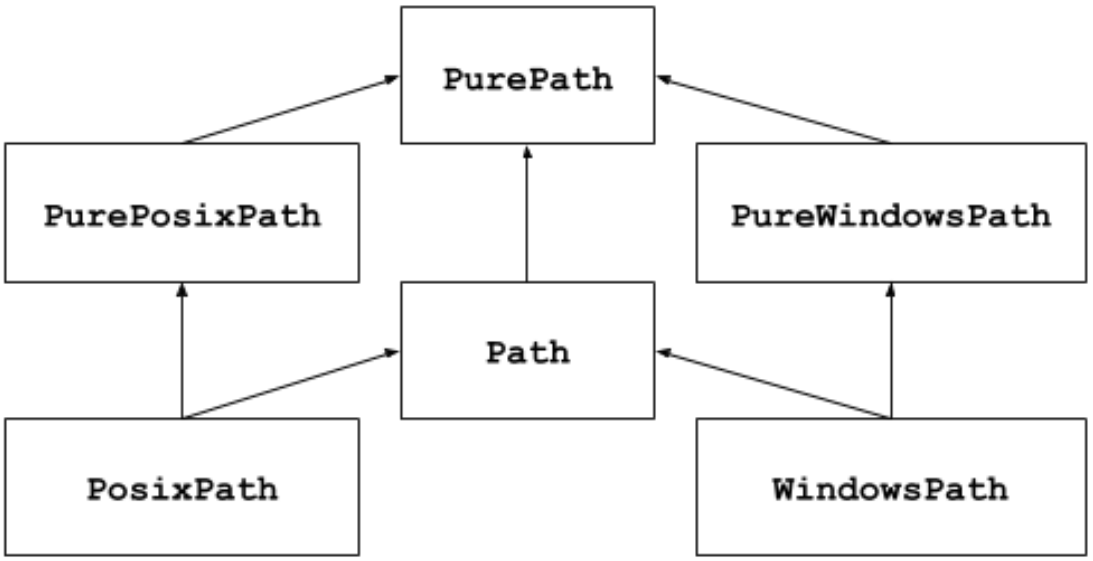
本章中描述的 module（模組）用於處理硬碟檔案和目F。例如，有一些 module 用於讀取檔案的屬性、以可F (portable) 方式操作路徑以及建立暫存檔。本章中的完整 module 清單是：

11.1 `pathlib` --- 物件導向檔案系統路徑

在 3.4 版被加入.

原始碼： Lib/pathlib/

此模組提供代表檔案系統路徑的類F，能適用不同作業系統的語意。路徑類F分成兩種，一種是純路徑 *(pure paths)*，提供F有 I/O 的單純計算操作，另一種是實體路徑 *(concrete paths)*，繼承自純路徑但也提供 IO 操作。



如果你之前從未使用過此模組或不確定哪個類🄴適合你的任務，那你需要的最有可能是*Path*。它針對程式執行所在的平台實例化一個實體路徑。

純路徑在某些特殊情境下是有用的，例如：

1. 如果你想在 Unix 機器上處理 Windows 路徑（或反過來），你無法在 Unix 上實例化*WindowsPath*，但你可以實例化*PureWindowsPath*。
2. 你想確保你的程式在操作路徑的時候不會真的存取到 OS。在這個情🄴下，實例化其中一種純路徑類🄴可能是有用的，因🄵它們不會有任何存取 OS 的操作。

也參考

PEP 428：pathlib 模組 -- 物件導向檔案系統路徑。

也參考

針對字串上的底層路徑操作，你也可以使用*os.path* 模組。

11.1.1 基本用法

匯入主要類🄵：

```
>>> from pathlib import Path
```

列出子目🄴：

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

在目前目🄴樹下列出 Python 原始碼檔案：

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

🄵覽目🄵樹🄵部：

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

查詢路徑屬性：

```
>>> q.exists()
True
>>> q.is_dir()
False
```

開🄴檔案：

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 例外

exception pathlib.**UnsupportedOperation**

繼承自*NotImplementedError* 的例外，當在路徑物件上呼叫不支援的操作時會被引發。

在 3.13 版被加入.

11.1.3 純路徑

純路徑物件提供處理路徑的操作，實際上不會存取檔案系統。有三種方式可以存取這些類🄵，我們也稱之🄵類型 *(flavours)*：

class pathlib.**PurePath**(**pathsegments*)

一個通用的類🄵，表示系統的路徑類型（實例化時會建立一個*PurePosixPath* 或*PureWindowsPath*）：

```
>>> PurePath('setup.py')      # 執行在 Unix 機器上
PurePosixPath('setup.py')
```

pathsegments 中的每個元素可以是以下的其中一種：一個表示路徑片段的字串，或一個物件，它實作了*os.PathLike* 介面且其中的*__fspath__()* 方法會回傳字串，就像是另一個路徑物件：

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

當🄵有給 *pathsegments* 的時候，會假設是目前的目🄵：

```
>>> PurePath()
PurePosixPath('.')
```

如果一個片段是🄴對路徑，則所有之前的片段會被忽略（類似*os.path.join()*）：

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

在 Windows 系統上，當遇到具有根目🄵的相對路徑片段（例如 `r'\foo'`）時，磁碟機 (drive) 部分不會被重置：

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

不必要的斜🄵和單點會被合🄵，但雙點 (`'..'`) 和前置的雙斜🄵 (`'//'`) 不會被合🄵，因🄵這樣會因🄵各種原因改變路徑的意義（例如符號連結 (symbolic links)、UNC 路徑）：

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

（一個使得 `PurePosixPath('foo/../bar')` 等同於 `PurePosixPath('bar')` 的單純方法，但如果 `foo` 是指到另一個目🄵的符號連結，就會是錯誤的。）

純路徑物件實作了*os.PathLike* 介面，使得它們可以在任何接受該介面的地方使用。

在 3.6 版的變更: 新增了對於*os.PathLike* 介面的支援。

class pathlib.**PurePosixPath**(**pathsegments*)

PurePath 的一個子類🄵，該路徑類型表示非 Windows 檔案系統的路徑：

```
>>> PurePosixPath('/etc/hosts')
PurePosixPath('/etc/hosts')
```

pathsegments 的指定方式與*PurePath* 類似。

class pathlib.**PureWindowsPath**(**pathsegments*)

PurePath 的一個子類🄵，該路徑類型表示 Windows 檔案系統的路徑，包括 UNC paths：

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

pathsegments 的指定方式與*PurePath* 類似。

不論你使用的是什🄵系統，你都可以實例化這些類🄵，因🄵它們不提供任何涉及系統呼叫的操作。

通用屬性

路徑物件是不可變 (immutable) 且可雜🄵 (*hashable*) 的。相同類型的路徑物件可以被比較和排序。這些屬性遵守該類型的大小寫語意規則：

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

不同類型的路徑物件在比較時視🄵不相等且無法被排序：

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
```

運算子

斜🄵運算子 (slash operator) 用於建立子路徑，就像是*os.path.join()* 函式一樣。如果引數是🄵對路徑，則忽略前一個路徑。在 Windows 系統上，當引數是具有根目🄵的相對路徑（例如，`r'\foo'`），磁碟機部分不會被重置：

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

路徑物件可以被用在任何可以接受實作*os.PathLike* 的物件的地方：

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

路徑的字串表示是原始的檔案系統路徑本身（以原生的形式，例如在 Windows 下是反斜🄵），你可以將其傳入任何將檔案路徑當作字串傳入的函式：

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

類似地，對路徑呼叫*bytes* 會得到原始檔案系統路徑的 bytes 物件，就像使用*os.fsencode()* 編碼過的一樣：

```
>>> bytes(p)
b'/etc'
```

備🄵

只建議在 Unix 下呼叫*bytes*。在 Windows 🄵，unicode 形式是檔案系統路徑的權威表示方式。

對個🄵組成的存取

可以使用下列屬性來存取路徑的個🄵「組成」(parts, components)：

PurePath.**parts**

一個可存取路徑的各組成的元組：

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

（特🄵注意磁碟機跟本地根目🄵是如何被重新組合成一個單一組成）

方法與屬性

純路徑提供以下方法與屬性：

PurePath.**parser**

用於底層路徑剖析和結合的*os.path* 模組的實作：可能是 posixpath 或 ntpath。

在 3.13 版被加入.

PurePath.**drive**

若存在則🄵一個表示磁碟機字母 (drive letter) 或磁碟機名稱 (drive name) 的字串：

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
```

(繼續下一頁)

(繼續上一頁)

```
>>> PurePosixPath('/etc').drive
''
```

UNC shares 也被視🄵磁碟機：

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**

若存在則🄵一個表示（本地或全域）根目🄵的字串：

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC shares 都會有一個根目🄵：

```
>>> PureWindowsPath('//host/share').root
'\\'
```

如果路徑以超過兩個連續的斜🄵開頭，*PurePosixPath* 會合🄵它們：

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

備🄵

此行🄵符合 *The Open Group Base Specifications Issue 6*，章節 4.11 路徑名稱解析：

「以兩個連續斜🄵開頭的路徑名懲可以根據實作定義的方式來解讀，🄵管如此，開頭超過兩個斜🄵應該視🄵單一斜🄵。」

PurePath.**anchor**

磁碟機與根目🄵的結合：

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

一個不可變的序列，🄵路徑邏輯上的祖先 (logical ancestors) 提供存取：

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
```

(繼續下一頁)

(繼續上一頁)

```
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

在 3.10 版的變更: 父序列現在支援*slices* 及負的索引值。

PurePath.**parent**

邏輯上的父路徑:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

你不能越過一個 anchor 或空路徑:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

備🄵

這是一個純粹字句上的 (lexical) 運算，因此會有以下行🄵:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

如果你想要沿任意的檔案系統路徑往上走，建議要先呼叫*Path.resolve()* 來解析符號連結 (symlink) 及去除其中的 ”..”。

PurePath.**name**

最後的路徑組成 (final path component) 的字串表示，不包含任何磁碟機或根目🄵:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC 磁碟機名稱🄵🄵有算在🄵:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**

以點分隔路徑的最後一個部分（如存在）:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

這通常被稱🄵檔案副檔名。

在 3.14 版的變更: A single dot (”.”) is considered a valid suffix.

PurePath.**suffixes**

一個路徑後綴 (suffix) 的串列，通常被稱🄵檔案副檔名：

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

在 3.14 版的變更: A single dot (”.”) is considered a valid suffix.

PurePath.**stem**

最後的路徑組成，不包括後綴：

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()

回傳一個使用正斜🄵 (/) 的路徑的字串表示：

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**is_absolute**()

回傳一個路徑是否是🄵對路徑。一個路徑被視🄵🄵對路徑的條件是它同時有根目🄵及（如果該系統類型允許的話）磁碟機：

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_relative_to**(*other*)

回傳此路徑是否🄵 *other* 路徑的相對路徑。

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

該方法是基於字串的；它既不存取檔案系統，也不特🄵處理”..” 片段。以下程式碼是等效的：

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

在 3.9 版被加入.

Deprecated since version 3.12, removed in version 3.14: 額外引數的傳入已🄵用；如果有的話，它們會與 *other* 連接在一起。

PurePath.**is_reserved**()

對*PureWindowsPath* 來🄵，當路徑在 Windows 下被視🄵保留的話會回傳 `True`，否則回傳 `False`。對*PurePosixPath* 來🄵，總是回傳 `False`。

在 3.13 版的變更: Windows 路徑名稱中包含冒號或結尾🄵點或空格會被視🄵保留。UNC 路徑可能被視🄵保留。

Deprecated since version 3.13, will be removed in version 3.15: 此方法已被🄵用；請使用*os.path.isreserved()* 來檢測 Windows 上的保留路徑。

PurePath.**joinpath**(**pathsegments*)

呼叫此方法會依序結合每個所給定的 *pathsegments* 到路徑上：

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**full_match**(*pattern*, *, *case_sensitive=None*)

將路徑與 glob 形式的模式 (glob-style pattern) 做比對。如果比對成功則回傳 `True`，否則回傳 `False`，例如：

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

也參考

模式語言 (pattern language) 文件。

像其它方法一樣，是否區分大小寫會遵循平台的預設行🄵：

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

將 *case_sensitive* 設定成 `True` 或 `False` 會覆蓋這個行🄵。

在 3.13 版被加入.

PurePath.**match**(*pattern*, *, *case_sensitive=None*)

將路徑與非遞🄵 glob 形式的模式 (glob-style pattern) 做比對。如果比對成功則回傳 `True`，否則回傳 `False`。

此方法類似於*full_match()*，但不允許空白模式（會引發*ValueError*）、不支援遞🄵萬用字元"**"（它會像非遞🄵的"*" 一樣），且如果提供相對模式，則會從右邊進行比對：

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

在 3.12 版的變更: *pattern* 參數接受類路徑物件。

在 3.12 版的變更: 新增 *case_sensitive* 參數。

PurePath.**relative_to**(*other*, *walk_up=False*)

計算這個路徑相對於 *other* 所表示路徑的版本。如果做不到會引發*ValueError*：

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is relative and␣
↪the other is absolute.
```

當 *walk_up* 是 False（預設值），路徑必須以 *other* 🄵開始。當此引數是 True，可能會加入 `..` 以組成相對路徑。在其他情🄵下，例如路徑參考到不同的磁碟機，則會引發*ValueError*：

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is relative and␣
↪the other is absolute.
```

⚠ 警告

這個函式是*PurePath* 的一部分且可以在字串上運作。它不會檢查或存取實際的檔案架構。這會影響到 *walk_up* 選項，因🄵它假設路徑中🄵有符號連結；如果需要解析符號連結的話可以先呼叫*resolve()*。

在 3.12 版的變更: 加入 *walk_up* 參數（舊的行🄵和 `walk_up=False` 相同）。

Deprecated since version 3.12, removed in version 3.14: 額外位置引數的傳入已🄵用；如果有的話，它們會與 *other* 連接在一起。

PurePath.**with_name**(*name*)

回傳一個修改*name* 後的新路徑。如果原始路徑🄵有名稱則引發 ValueError：

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_stem**(*stem*)

回傳一個修改*stem*後的新路徑。如果原始路徑🄵有名稱則引發 ValueError：

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

在 3.9 版被加入.

PurePath.**with_suffix**(*suffix*)

回傳一個修改*suffix*後的新路徑。如果原始路徑🄵有後綴，新的 *suffix* 會附加在後面。如果 *suffix* 是一個空字串，原來的後綴會被移除：

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

在 3.14 版的變更: A single dot (".") is considered a valid suffix. In previous versions, *ValueError* is raised if a single dot is supplied.

PurePath.**with_segments**(**pathsegments*)

透過結合給定的 *pathsegments* 建立一個相同型🄵的新路徑物件，當一個衍生路徑被建立的時候會呼叫這個方法，例如從*parent*和*relative_to()*建立衍生路徑。子類🄵可以覆寫此方法來傳遞資訊給衍生路徑，例如：

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id
```

(繼續下一頁)

(繼續上一頁)

```
    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

在 3.12 版被加入.

11.1.4 實體路徑

實體路徑是純路徑類🄴的子類🄴。除了後者本來就有提供的操作，它們也提供方法可以對路徑物件做系統呼叫。有三種方式可以實例化實體路徑：

class pathlib.**Path**(**pathsegments*)

PurePath 的子類🄴，此類🄴表示系統的路徑類型的實體路徑（實例化時會建立一個*PosixPath* 或*WindowsPath*）：

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments 的指定方式與*PurePath* 類似。

class pathlib.**PosixPath**(**pathsegments*)

Path 和*PurePosixPath* 的子類🄴，此類🄴表示實體非 Windows 檔案系統路徑：

```
>>> PosixPath('/etc/hosts')
PosixPath('/etc/hosts')
```

pathsegments 的指定方式與*PurePath* 類似。

在 3.13 版的變更: 在 Windows 上會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

class pathlib.**WindowsPath**(**pathsegments*)

Path 和*PureWindowsPath* 的子類🄴，此類🄴表示實體 Windows 檔案系統路徑：

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

pathsegments 的指定方式與*PurePath* 類似。

在 3.13 版的變更: 在非 Windows 平台上會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

你只能實例化對應你的系統的類🄴類型（允許在不相容的路徑類型上做系統呼叫可能在你的應用程式導致漏洞或故障）：

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

有些實體路徑方法會在系統呼叫失敗（例如因🄵路徑不存在）時引發*OSError*。

剖析和🄵生 URI

實體路徑物件可以從符合 **RFC 8089** 中的'file' URI 建立，也可以以該形式來表示。

備🄵

檔案 URI 在跨不同*檔案系統編碼*的機器上是無法移植的。

classmethod Path.**from_uri**(*uri*)

從剖析'file' URI 回傳新的路徑物件。例如：

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

在 Windows 上，從 URI 可以剖析 DOS 裝置和 UNC 路徑：

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

支援多種變形：

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

如果 URI 不是 `file:` 開頭，或是剖析後的路徑不是🄵對路徑，則會引發*ValueError*。

在 3.13 版被加入.

在 3.14 版的變更: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a *ValueError* is raised.

Path.**as_uri**()

以'file' URI 來表示路徑。如果路徑不是🄵對的則會引發*ValueError*。

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

Deprecated since version 3.14, will be removed in version 3.19: Calling this method from *PurePath* rather than *Path* is possible but deprecated. The method's use of *os.fsencode()* makes it strictly impure.

擴展和解析路徑

classmethod Path.**home**()

回傳一個代表使用者家目🄵的新的路徑物件（像以 ~ 構成的*os.path.expanduser()* 的回傳一樣）。如果無法解析家目🄵，會引發*RuntimeError*。

```
>>> Path.home()
PosixPath('/home/antoine')
```

在 3.5 版被加入.

Path.**expanduser**()

回傳一個展開 ~ 和 ~user 構成的新路徑，像*os.path.expanduser()* 回傳的一樣。如果無法解析家目🄵，會引發*RuntimeError*。

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

在 3.5 版被加入.

classmethod Path.**cwd**()

回傳一個代表目前目🄵的新的路徑物件（像*os.getcwd()* 回傳的一樣）：

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

Path.**absolute**()

將路徑轉🄵🄵🄵對路徑，不進行標準化或解析符號連結。回傳一個新的路徑物件：

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

Path.**resolve**(*strict=False*)

將路徑轉🄵🄵🄵對路徑，解析所有符號連結。回傳一個新的路徑物件：

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

同時也會消除”..” 的路徑組成（只有此方法這樣做）：

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

如果路徑不存在或遇到符號連結🄵圈，且 *strict* 🄵 `True`，則引發*OSError*。如果 *strict* 🄵 `False`，則將盡可能解析該路徑，🄵將任何剩余部分追加到路徑中，而不檢查其是否存在。

在 3.6 版的變更: 新增 *strict* 參數（在 3.6 版本之前的行🄵是嚴格的）。

在 3.13 版的變更: 在嚴格模式下，符號連結🄵圈會像其他錯誤一樣來處理：*OSError* 會被引發，而在非嚴格模式下，不會引發任何例外。在先前版本中，不管 *strict* 的值是什🄵，都會引發*RuntimeError*。

Path.**readlink**()

回傳符號連結指向的路徑（如*os.readlink()* 的回傳值）：

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

在 3.9 版被加入.

在 3.13 版的變更: 如果*os.readlink()* 不可用，會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

查詢檔案類型和狀態

在 3.8 版的變更: *exists()*、*is_dir()*、*is_file()*、*is_mount()*、*is_symlink()*、*is_block_device()*、*is_char_device()*、*is_fifo()*、*is_socket()* 遇到路徑包含 OS 層無法表示的字元時現在會回傳 `False` 而不是引發例外。

在 3.14 版的變更: The methods given above now return `False` instead of raising any *OSError* exception from the operating system. In previous versions, some kinds of *OSError* exception are raised, and others suppressed. The new behaviour is consistent with *os.path.exists()*, *os.path.isdir()*, etc. Use *stat()* to retrieve the file status without suppressing exceptions.

`Path.`**`stat`**`(*, `*follow_symlinks=True*`)`

回傳一個包含該路徑資訊的*os.stat_result* 物件，像*os.stat()* 一樣。每次呼叫此方法都會重新查詢結果。

此方法通常會跟隨 (follow) 符號連結；想要取得符號連結的資訊，可以加上引數 `follow_symlinks=False` 或使用*lstat()*。

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

在 3.10 版的變更: 新增 *follow_symlinks* 參數。

`Path.`**`lstat`**`()`

類似*Path.stat()*，但如果該路徑指向一個符號連結，則回傳符號連結的資訊而不是其指向的目標。

`Path.`**`exists`**`(*, `*follow_symlinks=True*`)`

Return `True` if the path points to an existing file or directory. `False` will be returned if the path is invalid, inaccessible or missing. Use *Path.stat()* to distinguish between these cases.

此方法通常會跟隨符號連結；如果想檢查符號連結是否存在，可以加上引數 `follow_symlinks=False`。

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

在 3.12 版的變更: 新增 *follow_symlinks* 參數。

`Path.`**`is_file`**`(*, `*follow_symlinks=True*`)`

Return `True` if the path points to a regular file. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a regular file. Use *Path.stat()* to distinguish between these cases.

此方法通常會跟隨符號連結；如果想將符號連結除外，可以加上引數 `follow_symlinks=False`。

在 3.13 版的變更: 新增 *follow_symlinks* 參數。

Path.**is_dir**(*, *follow_symlinks=True*)

Return `True` if the path points to a directory. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a directory. Use *Path.stat()* to distinguish between these cases.

此方法通常會跟隨符號連結；如果想將對目🄵的符號連結除外，可以加上引數 `follow_symlinks=False`。

在 3.13 版的變更: 新增 *follow_symlinks* 參數。

Path.**is_symlink**()

Return `True` if the path points to a symbolic link, even if that symlink is broken. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a symbolic link. Use *Path.stat()* to distinguish between these cases.

Path.**is_junction**()

如果該路徑指向一個連接點 (junction) 則回傳 `True`，對其他類型的檔案則回傳 `False`。目前只有 Windows 支援連接點。

在 3.12 版被加入.

Path.**is_mount**()

如果路徑是一個 *mount point*（一個檔案系統🄵🄵載不同檔案系統的存取點）則回傳 `True`。在 POSIX 上，此函式檢查 *path* 的父路徑 `path/..` 是否和 *path* 在不同的裝置上，或者 `path/..` 和 *path* 是否指向相同裝置的相同 i-node —— 這對於所有 Unix 和 POSIX 變體來🄵應該會偵測出🄵載點。在 Windows 上，一個🄵載點被視🄵一個根磁碟機字母（例如 `c:\`）、一個 UNC share（例如 `\\server\share`）或是🄵載的檔案系統目🄵。

在 3.7 版被加入.

在 3.12 版的變更: 加入對 Windows 的支援。

Path.**is_socket**()

Return `True` if the path points to a Unix socket. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a Unix socket. Use *Path.stat()* to distinguish between these cases.

Path.**is_fifo**()

Return `True` if the path points to a FIFO. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a FIFO. Use *Path.stat()* to distinguish between these cases.

Path.**is_block_device**()

Return `True` if the path points to a block device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a block device. Use *Path.stat()* to distinguish between these cases.

Path.**is_char_device**()

Return `True` if the path points to a character device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a character device. Use *Path.stat()* to distinguish between these cases.

Path.**samefile**(*other_path*)

回傳此路徑是否指向與 *other_path* 相同的檔案，*other_path* 可以是路徑 (Path) 物件或字串。其語義類似於*os.path.samefile()* 和*os.path.samestat()*。

若任何一個檔案因🄵某些原因無法存取，則引發*OSError*。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

在 3.5 版被加入.

Path.**info**

A *PathInfo* object that supports querying file type information. The object exposes methods that cache their results, which can help reduce the number of system calls needed when switching on file type. For example:

```
>>> p = Path('src')
>>> if p.info.is_symlink():
...     print('symlink')
... elif p.info.is_dir():
...     print('directory')
... elif p.info.exists():
...     print('something else')
... else:
...     print('not found')
...
directory
```

If the path was generated from *Path.iterdir()* then this attribute is initialized with some information about the file type gleaned from scanning the parent directory. Merely accessing *Path.info* does not perform any filesystem queries.

To fetch up-to-date information, it's best to call *Path.is_dir()*, *is_file()* and *is_symlink()* rather than methods of this attribute. There is no way to reset the cache; instead you can create a new path object with an empty info cache via `p = Path(p)`.

在 3.14 版被加入.

讀取和寫入檔案

Path.**open**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

開🄵該路徑指向的檔案，像🄵建的*open()* 函式做的一樣：

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.**read_text**(*encoding=None*, *errors=None*, *newline=None*)

將路徑指向的檔案的解碼🄵容以字串形式回傳：

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

該檔案被打開🄵且隨後關閉。可選參數的含義與*open()* 中的相同。

在 3.5 版被加入.

在 3.13 版的變更: 新增 *newline* 參數。

Path.**read_bytes**()

將路徑指向的檔案的二進位🄵容以一個位元組物件回傳：

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

在 3.5 版被加入.

Path.**write_text**(*data*, *encoding=None*, *errors=None*, *newline=None*)

以文字模式開🄴指向的檔案，將 *data* 寫到檔案，🄴關閉檔案::：

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

一個名稱相同的已存在檔案會被覆寫。可選參數和*open()* 的參數有相同意義。

在 3.5 版被加入.

在 3.10 版的變更: 新增 *newline* 參數。

Path.**write_bytes**(*data*)

以位元組模式開🄴指向的檔案，將 *data* 寫到檔案，🄴關閉檔案::：

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

一個名稱相同的已存在檔案會被覆寫。

在 3.5 版被加入.

讀取目🄵

Path.**iterdir**()

當該路徑指向一個目🄵，會 yield 目🄵🄵面的路徑物件：

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

子路徑會以任意順序被 yield，且不會包含特殊項目 `'.'` 和 `'..'`。如果一個檔案在建立這個🄴代器之後加到該目🄵或從目🄵🄵除，這個檔案的路徑物件是否會被包含是🄴有明定的。

如果路徑不是目🄵或無法存取，則會引發*OSError*。

Path.**glob**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

在該路徑表示的目🄵🄵，以 glob 方式比對所給定的相對 *pattern*，🄴 yield 所有比對到的檔案（任意類型）：

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

> **也參考**
>
> 模式語言 *(pattern language)* 文件。

預設情🄵下，或者當 *case_sensitive* 僅限關鍵字引數被設定🄵 `None` 的時候，此方法會使用平台特定的大小寫規則來比對路徑；通常在 POSIX 上會區分大小寫，而在 Windows 上不區分大小寫。將 *case_sensitive* 設成 `True` 或 `False` 會覆寫這個行🄵。

預設情🄵下，或者當 *recurse_symlinks* 僅限關鍵字引數被設定🄵 `False` 的時候，此方法會跟隨符號連結，除非在擴展"`**`" 萬用字元時。將 *recurse_symlinks* 設成 `True` 以總是跟隨符號連結。

引發一個附帶引數 `self`、`pattern` 的稽核事件 `pathlib.Path.glob`。

在 3.12 版的變更: 新增 *case_sensitive* 參數。

在 3.13 版的變更: 新增 *recurse_symlinks* 參數。

在 3.13 版的變更: *pattern* 參數接受類路徑物件。

在 3.13 版的變更: 從掃描檔案系統引發的任何*OSError* 例外都會被抑制。在先前版本中，在許多情🄵下這種例外都會被抑制，但不是所有情🄵。

`Path.`**`rglob`**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

遞🄵地 glob 給定的相對 *pattern*。這相當於在給定的相對 *pattern* 前面加上"`**/`" 🄵呼叫*Path.glob()*。

> **也參考**
>
> 模式語言 *(pattern language)* 和*Path.glob()* 文件。

引發一個附帶引數 `self`、`pattern` 的稽核事件 `pathlib.Path.rglob`。

在 3.12 版的變更: 新增 *case_sensitive* 參數。

在 3.13 版的變更: 新增 *recurse_symlinks* 參數。

在 3.13 版的變更: *pattern* 參數接受類路徑物件。

`Path.`**`walk`**(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

透過由上而下或由下而上地走訪目🄵樹🄵生目🄵樹🄵的檔案名稱。

對每個以 *self* 🄵根且在目🄵樹🄵的目🄵（包含 *self* 但不包含'.' 和'..' ），此方法會 yield 一個 `(dirpath, dirnames, filenames)` 的三元素元組。

dirpath 是一個目前走訪到的目🄵的*Path*，*dirnames* 是一個 *dirpath* 🄵的子目🄵名稱的字串串列（不包含 `'.'` 和 `'..'`），而 *filenames* 是一個 *dirpath* 🄵非目🄵檔案名稱的字串串列。要取得在 *dirpath* 🄵檔案或目🄵的完整路徑（以 *self* 開頭），可以使用 `dirpath / name`。會根據檔案系統來🄵定串列是否有排序。

如果可選引數 *top_down* 是 true（預設值），一個目🄵的三元素元組會在其任何子目🄵的三元素元組之前🄵生（目🄵是由上而下走訪）。如果 *top_down* 是 false，一個目🄵的三元素元組會在其所有子目🄵的三元素元組之後🄵生（目🄵是由下而上走訪）。不論 *top_down* 的值是什🄵，子目🄵的串列會在走訪該目🄵及其子目🄵的三元素元組之前取得。

當 *top_down* 是 true，呼叫者可以原地 (in-place) 修改 *dirnames* 串列（例如使用 `del` 或切片賦值 (slice assignment)），且*Path.walk()* 只會遞🄵進名稱依然留在 *dirnames* 🄵的子目🄵。這可以用來修剪搜尋，或者🄵加特定順序的🄵覽，或者甚至在繼續*Path.walk()* 之前，用來告訴*Path.walk()* 關於呼叫者建立或重新命名的目🄵。當 *top_down* 是 false 的時候，修改 *dirnames* 對*Path.walk()* 的行🄵🄵有影響，因🄵 *dirnames* 🄵的目🄵已經在 *dirnames* yield 給呼叫者之前被🄵生。

預設來自*os.scandir()* 的錯誤會被忽略。如果指定了可選引數 *on_error*（它應該要是一個可呼叫物件），它會被以一個*OSError* 實例🄵引數來呼叫。這個可呼叫物件可以處理錯誤以繼續走訪，或者再次引發錯誤來停止走訪。注意，檔案名稱可以從例外物件的 `filename` 屬性來取得。

預設*Path.walk()* 不會跟隨符號連結，而是會把它們加到 *filenames* 串列。將 *follow_symlinks* 設定🄵 true 會解析符號連結，🄵將它們根據其指向的目標放在適當的 *dirnames* 和 *filenames*，而因此🄵覽到符號連結指向的目🄵（在有支援符號連結的地方）。

備🄵

需要注意的是如果符號連結指向一個其本身的父目🄵，則將 *follow_symlinks* 設定🄵 true 會導致無窮的遞🄵。*Path.walk()* 不會紀🄵其已經🄵覽過的目🄵。

備🄵

Path.walk() 假設其走訪的目🄵在執行過程中不會被修改。舉例來🄵，如果在 *dirnames* 🄵的目🄵已經被一個符號連結取代，且 *follow_symlinks* 是 false，*Path.walk()* 依然會試著往下進入它。🄵了防止這樣的行🄵，可以從 *dirnames* 適當地移除目🄵。

備🄵

如果 *follow_symlinks* 是 false，和*os.walk()* 行🄵不同的是*Path.walk()* 會將指向目🄵的符號連結放在 *filenames* 串列。

這個範例會顯示在每個目🄵🄵所有檔案使用的位元組數量，同時間忽略 __pycache__ 目🄵：

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

下一個範例是*shutil.rmtree()* 的一個簡單的實作方式。由下而上走訪目🄵樹是必要的，因🄵*rmdir()* 不允許在目🄵🄵空之前🄵除它：

```
# 🄵除可從 "top" 目🄵到達的所有東西。
# 注意：這是危險的！例如，如果 top == Path('/')，
# 它可能會🄵除你所有的檔案。
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

在 3.12 版被加入.

建立檔案和目🄵

Path.**touch**(*mode=0o666*, *exist_ok=True*)

根據給定路徑來建立一個檔案。如果 *mode* 有給定，它會與行程的 umask 值結合，以確定檔案模式和存取旗標。當檔案已經存在時，若 *exist_ok* 🄵 true 則函式不會失敗（其變更時間會被更新🄵當下時間），否則會引發*FileExistsError*。

> **也參考**
>
> *open()*、*write_text()* 和*write_bytes()* 方法通常用於建立檔案。

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)

在給定路徑下建立一個新的目🄵。如果有給 *mode* 則會結合行程 (process) 的 `umask` 值來🄵定檔案模式與存取旗標 (access flag)。如果路徑已經存在，會引發*FileExistsError*。

如果 *parents* 是 true，則任何缺少的父路徑都會依需要被建立；它們不考慮 *mode* 而會以預設的權限來建立（模仿 POSIX 的 `mkdir -p` 指令）。

如果 *parents* 是 false（預設值），缺少的父路徑會引發*FileNotFoundError*。

如果 *exist_ok* 是 false（預設值），則當目標目🄵已經存在的話會引發*FileExistsError*。

如果 *exist_ok* 是 true，只有當最後的路徑組成不是一個已存在的非目🄵檔案，*FileExistsError* 例外會被忽略（與 POSIX 的 `mkdir -p` 指令行🄵相同）。

在 3.5 版的變更: 新增 *exist_ok* 參數。

Path.**symlink_to**(*target*, *target_is_directory=False*)

使這個路徑成🄵一個指向 *target* 的符號連結。

在 Windows 上，符號連結代表一個檔案或目🄵，且不會隨著目標 (target) 動態改變。如果目標存在，則符號連結的類型會被建立來符合其目標。否則如果 *target_is_directory* 是 true，該符號連結會被建立成目🄵，如果不是則建立成檔案（預設值）。在非 Windows 平台上，*target_is_directory* 會被忽略。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

> **備🄵**
>
> 引數的順序 (link, target) 和*os.symlink()* 相反。

在 3.13 版的變更: 如果*os.symlink()* 不可用，會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

Path.**hardlink_to**(*target*)

使這個路徑成🄵與 *target* 相同檔案的一個硬連結 (hard link)。

> **備🄵**
>
> 引數的順序 (link, target) 和*os.link()* 相反。

在 3.10 版被加入.

在 3.13 版的變更: 如果*os.link()* 不可用，會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

FF、移動和F除

Path.**copy**(*target*, *, *follow_symlinks=True*, *preserve_metadata=False*)

將此檔案或目F樹FF到給定的 *target*，F回傳一個指向 *target* 的新 Path 實例。

If the source is a file, the target will be replaced if it is an existing file. If the source is a symlink and *follow_symlinks* is true (the default), the symlink's target is copied. Otherwise, the symlink is recreated at the destination.

If *preserve_metadata* is false (the default), only directory structures and file data are guaranteed to be copied. Set *preserve_metadata* to true to ensure that file and directory permissions, flags, last access and modification times, and extended attributes are copied where supported. This argument has no effect when copying files on Windows (where metadata is always preserved).

> **備F**
>
> Where supported by the operating system and file system, this method performs a lightweight copy, where data blocks are only copied when modified. This is known as copy-on-write.

在 3.14 版被加入.

Path.**copy_into**(*target_dir*, *, *follow_symlinks=True*, *preserve_metadata=False*)

Copy this file or directory tree into the given *target_dir*, which should be an existing directory. Other arguments are handled identically to *Path.copy()*. Returns a new Path instance pointing to the copy.

在 3.14 版被加入.

Path.**rename**(*target*)

將此檔案或目F重新命名F給定的 *target* ，F回傳一個新的 Path 實例指向該 *target*。在 Unix 系統上，若 *target* 存在且F一個檔案，若使用者有權限，則會在不顯示訊息的情F下進行取代。在 Windows 系統上，若 *target* 存在，則會引發*FileExistsError* 錯誤。*target* 可以是字串或另一個路徑物件:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

目標路徑可以是F對路徑或相對路徑。相對路徑會相對於目前的工作目F進行解釋，而不是相對於 Path 物件所在的目F。

此功能是使用*os.rename()* 實現的，F提供相同的保證。

在 3.8 版的變更: 新增了回傳值，回傳新的 Path 實例。

Path.**replace**(*target*)

將此檔案或目F重新命名F給定的 *target*，F回傳一個指向 *target* 的新 Path 實例。如果 *target* 指向一個現有的檔案或空目F，它將被無條件地取代。

目標路徑可以是F對路徑或相對路徑。相對路徑會相對於目前的工作目F進行解釋，而不是相對於 Path 物件所在的目F。

在 3.8 版的變更: 新增了回傳值，回傳新的 Path 實例。

Path.**move**(*target*)

將此檔案或目F樹移動到給定的 *target*，F回傳一個指向 *target* 的新 Path 實例。

If the *target* doesn't exist it will be created. If both this path and the *target* are existing files, then the target is overwritten. If both paths point to the same file or directory, or the *target* is a non-empty directory, then *OSError* is raised.

If both paths are on the same filesystem, the move is performed with *os.replace()*. Otherwise, this path is copied (preserving metadata and symlinks) and then deleted.

在 3.14 版被加入.

Path.**move_into**(*target_dir*)

Move this file or directory tree into the given *target_dir*, which should be an existing directory. Returns a new Path instance pointing to the moved path.

在 3.14 版被加入.

Path.**unlink**(*missing_ok=False*)

移除這個檔案或符號連結。如果路徑指向目🄵，請改用*Path.rmdir()*。

如果 *missing_ok* 是 false（預設值），*FileNotFoundError* 會在路徑不存在時被引發。

如果 *missing_ok* 是 true，*FileNotFoundError* 例外會被忽略（行🄵與 POSIX `rm -f` 指令相同）。

在 3.8 版的變更: 新增 *missing_ok* 參數。

Path.**rmdir**()

移除此目🄵。該目🄵必須🄵空。

權限和所有權

Path.**owner**(*, *follow_symlinks=True*)

回傳擁有該檔案的用🄵名稱。如果在系統資料庫中找不到該檔案的使用者識🄵字 (UID)，則會引發*KeyError*。

此方法通常會跟隨符號連結；如果想取得符號連結的擁有者，可以加上引數 `follow_symlinks=False`。

在 3.13 版的變更: 如果*pwd* 模組不可用，會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

在 3.13 版的變更: 新增 *follow_symlinks* 參數。

Path.**group**(*, *follow_symlinks=True*)

回傳擁有該檔案的群組名稱。如果在系統資料庫🄵找不到檔案的群組識🄵字 (GID) 會引發*KeyError*。

此方法通常會跟隨符號連結；如果想取得符號連結的群組，可以加上引數 `follow_symlinks=False`。

在 3.13 版的變更: 如果*grp* 模組不可用，會引發*UnsupportedOperation*。在先前版本中，則是引發*NotImplementedError*。

在 3.13 版的變更: 新增 *follow_symlinks* 參數。

Path.**chmod**(*mode*, *, *follow_symlinks=True*)

修改檔案模式 (file mode) 與權限，像*os.chmod()* 一樣。

此方法通常會跟隨符號連結。一些 Unix 類型支援修改符號連結本身的權限；在這些平台上你可以加上引數 `follow_symlinks=False` 或使用*lchmod()*。

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

在 3.10 版的變更: 新增 *follow_symlinks* 參數。

Path.**lchmod**(*mode*)

類似*Path.chmod()*，但如果該路徑指向一個符號連結，則符號連結的模式 (mode) 會被改變而不是其指向的目標。

11.1.5 模式語言 (pattern language)

以下的萬用字元在*full_match()*、*glob()* 和*rglob()* 的模式中被支援：

****（整個片段）**
匹配任何數量的檔案或目🄵片段，包括零個。

***（整個片段）**
匹配一個檔案或目🄵的片段。

***（片段的一部分）**
匹配任意數量的非分隔字元，包括零個。

?
匹配一個非分隔字元。

[seq]
Matches one character in *seq*, where *seq* is a sequence of characters. Range expressions are supported; for example, `[a-z]` matches any lowercase ASCII letter. Multiple ranges can be combined: `[a-zA-Z0-9_]` matches any ASCII letter, digit, or underscore.

[!seq]
匹配一個不在 *seq* 中的字元，其中 *seq* 遵循上述相同的規則。

對於文本 (literal) 匹配，可以用方括號包裝元字元 (meta-characters)。例如，`"[?]"` 會匹配字元 `"?"`。

"`**`" 萬用字元讓它可以做遞🄵 glob。例如：

模式	意涵
"`**/*`"	至少有一個片段的路徑。
"`**/*.py`"	最後一個片段以"`.py`" 結尾的任何路徑。
"`assets/**`"	任何以"`assets/`" 開頭的路徑。
"`assets/**/*`"	任何以"`assets/`" 開頭的路徑，不包括"`assets/`" 本身。

備🄵

Glob 使用 `**` 萬用字元會訪問目🄵樹中的每個目🄵。對於大型的目🄵樹，搜尋可能需要花費很長的時間。

在 3.13 版的變更: Glob 使用結尾🄵 `**` 的模式會同時回傳檔案和目🄵。在先前版本中，只會回傳目🄵。

在*Path.glob()* 和*rglob()* 中，可以在模式後面加上斜🄵以只匹配目🄵。

在 3.11 版的變更: Glob 使用以路徑名稱組成的分隔符號（*sep* 或*altsep*）作結尾的模式則只會回傳目🄵。

11.1.6 與 `glob` 模組的比較

Path.glob() 和*Path.rglob()* 接受的模式和🄵生的結果與*glob* 模組略有不同：

1. pathlib 中以點開頭的檔案🄵不特🄵。這和將 `include_hidden=True` 傳遞給*glob.glob()* 相同。
2. `**` 模式組成在 pathlib 中總是遞🄵的。這與將 `recursive=True` 傳遞給*glob.glob()* 相同。
3. 在 pathlib 中，`**` 模式組成預設不跟隨符號連結。這在*glob.glob()* 中🄵有等效的行🄵，但你可以將 `recurse_symlinks=True` 傳遞給*Path.glob()* 以獲得相容的行🄵。
4. 與所有*PurePath* 和*Path* 物件一樣，從*Path.glob()* 和*Path.rglob()* 回傳的值不包含結尾斜🄵。
5. pathlib 的 `path.glob()` 和 `path.rglob()` 回傳的值包含了 *path* 作🄵前綴，而 `glob.glob(root_dir=path)` 的結果則不會如此。
6. pathlib 的 `path.glob()` 和 `path.rglob()` 回傳的值可能包含 *path* 本身，例如當使用"`**`" 做 glob 的時候，然而 `glob.glob(root_dir=path)` 的結果則永遠不會包含一個對應到 *path* 的空字串。

11.1.7 與 os 和 os.path 模組的比較

pathlib 使用*PurePath* 和*Path* 物件來實作路徑操作，因此它被稱F是物件導向的。另一方面，*os* 和*os.path* 模組提供能F與底層 str 和 bytes 物件互動的函式，這是一種更程序式的方法。有些使用者認F物件導向的風格更易讀。

os 和*os.path* 中的許多函式支援 bytes 路徑和*相對路徑的目F描述器 (paths relative to directory descriptors)*。這些功能在 pathlib 中不可用。

Python 的 str 和 bytes 型F，以及*os* 和*os.path* 模組的一些部分，是用 C 寫的且非常快速。pathlib 是用純 Python 寫的且通常比較慢，但很少會慢到足以F生影響。

pathlib 的路徑正規化略比*os.path* 更武斷和一致。例如，F管*os.path.abspath()* 會從路徑中移除".." 片段，如果包含符號連結的話這可能會改變其意義，而*Path.absolute()* 則會保留這些片段以增加安全性。

pathlib 的路徑正規化可能會使它不適合某些應用程式：

1. pathlib 將 Path("my_folder/") 正規化F Path("my_folder")，這會在提供給各種操作系統 API 和命令列工具時改變路徑的意義。具體來F，缺少結尾分隔符號可能會允許該路徑被解析F檔案或目F，而不只是目F。
2. pathlib 將 Path("./my_program") 正規化F Path("my_program")，這會在作F執行檔搜尋路徑使用時改變路徑的意義，例如在 shell 或在衍生 (spawn) 子行程時。具體來F，在路徑中缺少分隔符號可能會使其F制在 PATH 中尋找，而不是目前目F。

因F這些差F，pathlib 不是*os.path* 的直接替代品。

對應工具

以下是一張表格，對應許多*os* 函式及其相符於*PurePath*/*Path* 的項目。

os 和*os.path*	*pathlib*
os.path.dirname()	*PurePath.parent*
os.path.basename()	*PurePath.name*
os.path.splitext()	*PurePath.stem* 和*PurePath.suffix*
os.path.join()	*PurePath.joinpath()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.relpath()	*PurePath.relative_to()*[1]
os.path.expanduser()	*Path.expanduser()*[2]
os.path.realpath()	*Path.resolve()*
os.path.abspath()	*Path.absolute()*[3]
os.path.exists()	*Path.exists()*
os.path.isfile()	*Path.is_file()*
os.path.isdir()	*Path.is_dir()*
os.path.islink()	*Path.is_symlink()*
os.path.isjunction()	*Path.is_junction()*
os.path.ismount()	*Path.is_mount()*
os.path.samefile()	*Path.samefile()*
os.getcwd()	*Path.cwd()*
os.stat()	*Path.stat()*
os.lstat()	*Path.lstat()*
os.listdir()	*Path.iterdir()*
os.walk()	*Path.walk()*[4]
os.mkdir()、*os.makedirs()*	*Path.mkdir()*
os.link()	*Path.hardlink_to()*
os.symlink()	*Path.symlink_to()*
os.readlink()	*Path.readlink()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*

繼續下一頁

表格 1 – 繼續上一頁

os 和os.path	pathlib
os.remove()、os.unlink()	Path.unlink()
os.rmdir()	Path.rmdir()
os.chmod()	Path.chmod()
os.lchmod()	Path.lchmod()

11.1.8 Protocols

The *pathlib.types* module provides types for static type checking.

在 3.14 版被加入.

class pathlib.types.**PathInfo**

A *typing.Protocol* describing the *Path.info* attribute. Implementations may return cached results from their methods.

exists(*, *follow_symlinks=True*)

如果路徑是個存在的檔案、目🄵或任何其他類型的檔案則回傳 `True`，如果路徑不存在則回傳 `False`。

If *follow_symlinks* is `False`, return `True` for symlinks without checking if their targets exist.

is_dir(*, *follow_symlinks=True*)

Return `True` if the path is a directory, or a symbolic link pointing to a directory; return `False` if the path is (or points to) any other kind of file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a directory (without following symlinks); return `False` if the path is any other kind of file, or if it doesn't exist.

is_file(*, *follow_symlinks=True*)

Return `True` if the path is a file, or a symbolic link pointing to a file; return `False` if the path is (or points to) a directory or other non-file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a file (without following symlinks); return `False` if the path is a directory or other non-file, or if it doesn't exist.

is_symlink()

Return `True` if the path is a symbolic link (even if broken); return `False` if the path is a directory or any kind of file, or if it doesn't exist.

11.2 os.path --- 常見的路徑名操作

原始碼： Lib/genericpath.py、Lib/posixpath.py (用於 POSIX 系統) 和 Lib/ntpath.py (用於 Windows).

該模組實現了一些有用的路徑名操作函式。若要讀取或寫入檔案，請參🄵*open()* 函式，要存取檔案系統，請參🄵*os* 模組。路徑參數可以以字串、位元組或任何依照*os.PathLike* 協定實作的物件傳遞。

與 Unix shell 不同，Python 不會自動進行路徑展開 (path expansions)。當應用程式需要進行類似 shell 的路徑展開時，可以明確地叫用*expanduser()* 和*expandvars()* 等函式。(另請參🄵*glob* 模組。)

[1] *os.path.relpath()* 會呼叫*abspath()* 來將路徑變成🄵對路徑🄵移除".." 部分，而*PurePath.relative_to()* 是一個文本上的操作，當它輸入的錨點不同時（例如一個是🄵對路徑，另一個則是相對路徑）會引發*ValueError*。

[2] *os.path.expanduser()* 會在無法解析家目🄵時回傳原始路徑，而*Path.expanduser()* 則會引發*RuntimeError*。

[3] *os.path.abspath()* 將".." 組成移除而不解析符號連結，這可能會改變路徑的意義，而*Path.absolute()* 則會保留路徑中任何".." 組成。

[4] 當分類路徑成 *dirnames* 和 *filenames* 時*os.walk()* 總是跟隨符號連結，而*Path.walk()* 在 *follow_symlinks* 🄵 false（預設值）時，會將所有符號連結都分類🄵 *filenames*。

也參考

pathlib 模組提供了高階的路徑物件。

備🄵

所有這些函式都只接受位元組或字串物件作🄵參數。如果回傳的是路徑或檔案名稱，結果將是相同型🄵的物件。

備🄵

由於不同的作業系統具有不同的路徑命名慣例，在標準函式庫中的路徑模組有數個版本可供使用，而*os.path* 模組都會是運行 Python 之作業系統所適用本地路徑。然而，如果你想要操作始終以某個不同於本機格式表示的路徑，你也可以引入🄵使用對應的模組。它們都具有相同的介面：

- `posixpath` 用於 UNIX 形式的路徑
- `ntpath` 用於 Windows 的路徑

在 3.8 版的變更: 對於包含有作業系統層級無法表示之字元或位元組的路徑，*exists()*、*lexists()*、*isdir()*、*isfile()*、*islink()* 和*ismount()* 函式現在會回傳 `False`，而不是引發例外。

`os.path.`**`abspath`**(*path*)

回傳經正規化的🄵對路徑名 *path* 。在大多數平台上，這等效於按照以下方式呼叫*normpath()* 函式：`normpath(join(os.getcwd(), path))`。

在 3.6 版的變更: 接受一個*path-like object*。

`os.path.`**`basename`**(*path*, */*)

回傳路徑名 *path* 的基底名稱。這是將 *path* 傳遞給函式*split()* 後回傳結果中的第二個元素。請注意，此函式的結果與 Unix 的 **basename** 程式不同；對於 `'/foo/bar/'`，**basename** 回傳 `'bar'`，而*basename()* 函式回傳空字串（`''`）。

在 3.6 版的變更: 接受一個*path-like object*。

`os.path.`**`commonpath`**(*paths*)

回傳可🄵代物件 *paths* 中每個路徑名稱的最長共同子路徑。如果 *paths* 同時包含🄵對路徑名稱和相對路徑名稱、*paths* 位於不同的驅動機或 *paths* 🄵空，則引發*ValueError*。與*commonprefix()* 不同，此函式回傳的是有效路徑。

在 3.5 版被加入.

在 3.6 版的變更: 接受一個類路徑物件的序列。

在 3.13 版的變更: Any iterable can now be passed, rather than just sequences.

`os.path.`**`commonprefix`**(*list*, */*)

回傳 *list* 中所有路徑的最長路徑前綴（逐字元比較）。如果 *list* 🄵空，則回傳空字串（`''`）。

備🄵

由於此函式是逐字元比較，因此可能會回傳無效的路徑。若要獲得有效的路徑，請參考*commonpath()* 函式。

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**dirname**(*path*, /)

回傳路徑名 *path* 的目🄵名稱。這是將 *path* 傳遞給函式*split()* 後回傳之成對結果中的第一個元素。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**exists**(*path*)

如果 *path* 是一個存在的路徑或一個開🄵的檔案描述器則回傳 `True`。對於已損壞的符號連結則回傳 `False`。在某些平台上，即使 *path* 實際存在，如果未被授予執行*os.stat()* 的權限，此函式仍可能回傳 `False`。

在 3.3 版的變更: 現在 *path* 可以是一個整數：如果它是一個開🄵的檔案描述器，則回傳 `True`；否則回傳 `False`。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**lexists**(*path*)

如果 *path* 是一個存在的路徑則回傳 `True`，對已損壞的符號連結也是。在缺乏*os.lstat()* 的平台上，與*exists()* 函式等效。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**expanduser**(*path*)

在 Unix 和 Windows 上，將引數中以 `~` 或 `~user` 開頭的部分替🄵🄵該 *user* 的家目🄵。

在 Unix 上，如果環境變數 `HOME` 有被設置，則將初始的 `~` 替🄵🄵該變數的值；否則將使用🄵建模組*pwd* 在密碼目🄵中查找目前使用者的家目🄵。對於初始的 `~user`，直接在密碼目🄵中查找該使用者的家目🄵。

在 Windows 上，如果 `USERPROFILE` 有被設置，則使用該變數的值；否則將結合 `HOMEPATH` 和 `HOMEDRIVE`。對於初始的 `~user`，會檢查目前使用者的家目🄵的最後一個目🄵元件是否與 `USERNAME` 相符，如果相符則替🄵它。

如果展開失敗或路徑不以波浪符號 (tilde) 開頭，則回傳原始路徑，不做任何變更。

在 3.6 版的變更: 接受一個*path-like object*。

在 3.8 版的變更: 在 Windows 上不再使用 `HOME` 變數。

os.path.**expandvars**(*path*)

回傳已展開環境變數的引數。形如 `$name` 或 `${name}` 的子字串會被替🄵🄵環境變數 *name* 的值。無效的變數名稱和對不存在變數的引用保持不變。

在 Windows 上，除了支援 `$name` 和 `${name}` 形式的展開外，還支援 `%name%` 形式的展開。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**getatime**(*path*, /)

回傳 *path* 的最後存取時間。回傳值是一個浮點數，表示自紀元（參見*time* 模組）以來的秒數。如果檔案不存在或無法存取，則引發*OSError*。

os.path.**getmtime**(*path*, /)

回傳 *path* 的最後修改時間。回傳值是一個浮點數，表示自紀元（參見*time* 模組）以來的秒數。如果檔案不存在或無法存取，則引發*OSError*。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**getctime**(*path*, /)

回傳系統的 ctime。在某些系統（如 Unix）上，這是最後一次元數據 (metadata) 更改的時間，在其他系統（如 Windows）上則是 *path* 的建立時間。回傳值是一個浮點數，表示自紀元（參見*time* 模組）以來的秒數。如果檔案不存在或無法存取），則引發*OSError*。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**getsize**(*path*, /)

回傳 *path* 的大小（以位元組🄵單位）。如果檔案不存在或無法存取，則引發*OSError*。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**isabs**(*path*, /)

如果 *path* 是🄵對路徑名，則回傳 True。在 Unix 上，這表示它以斜🄵開頭；在 Windows 上，表示這是以兩個（反）斜🄵開頭或者以一個驅動機字母、冒號和（反）斜🄵開頭。

在 3.6 版的變更: 接受一個*path-like object*。

在 3.13 版的變更: On Windows, returns False if the given path starts with exactly one (back)slash.

os.path.**isfile**(*path*)

如果 *path* 是一個*已存在的*常規檔案，則回傳 True。這將跟隨符號連結，因此同一個路徑可以同時回傳*islink()* 和*isfile()* 的結果🄵真。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**isdir**(*path*, /)

如果 *path* 是一個*已存在的*目🄵，則回傳 True。這將跟隨符號連結，因此同一個路徑可以同時回傳*islink()* 和*isfile()* 的結果🄵真。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**isjunction**(*path*)

如果 *path* 是指向*已存在的*目🄵條目且🄵聯接點 (junction)，則回傳 True。如果目前平台不支援聯接點，則始終返回 False。

在 3.12 版被加入.

os.path.**islink**(*path*)

如果 *path* 是指向*已存在的*目🄵項目且🄵符號連結，則回傳 True。如果 Python 執行時不支援符號連結，則始終回傳 False。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**ismount**(*path*)

如果路徑名 *path* 是一個🄵載點 (*mount point*)，則回傳 True：即在檔案系統中🄵載了不同的檔案系統。在 POSIX 系統上，該函式檢查 *path* 的父目🄵 *path*/.. 是否位於不同的設備上，或者 *path*/.. 和 *path* 是否指向同一設備上的相同 i-node --- 這應該能🄵檢測出所有 Unix 和 POSIX 變體的🄵載點。但無法可靠地檢測出相同檔案系統上的綁定🄵載點 (bind mount)。在 Linux 系統上，即使它們不是🄵載點，也會始終回傳 True 給 btrfs 子卷（subvolumes）。在 Windows 上，以驅動機字母開頭的根目🄵和 UNC 共享路徑始終是🄵載點，對於任何其他路徑，會呼叫 GetVolumePathName 函式來檢查它是否與輸入路徑不同。

在 3.4 版的變更: 新增在 Windows 上檢測非根目🄵🄵載點的支援。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**isdevdrive**(*path*)

如果路徑名 *path* 位於 Windows Dev 驅動機上，則回傳 True。Dev 驅動機針對開發人員場景進行了優化，提供更快的讀寫檔案性能。建議將其用於原始程式碼、臨時建置目🄵、封包快取和其他 I/O 密集型操作。

可能會對無效的路徑引發錯誤，例如，🄵有可識🄵的驅動機的路徑，但在不支援 Dev 磁碟機的平台上返回 False。請參🄵 Windows 文件以了解有關🄵用和建立 Dev 驅動機的資訊。

在 3.12 版被加入.

在 3.13 版的變更: The function is now available on all platforms, and will always return False on those that have no support for Dev Drives

os.path.**isreserved**(*path*)

Return `True` if *path* is a reserved pathname on the current system.

On Windows, reserved filenames include those that end with a space or dot; those that contain colons (i.e. file streams such as "name:stream"), wildcard characters (i.e. `'*?"<>'`), pipe, or ASCII control characters; as well as DOS device names such as "NUL", "CON", "CONIN$", "CONOUT$", "AUX", "PRN", "COM1", and "LPT1".

> **備🄵**
>
> This function approximates rules for reserved paths on most Windows systems. These rules change over time in various Windows releases. This function may be updated in future Python releases as changes to the rules become broadly available.

可用性: Windows.

在 3.13 版被加入.

os.path.**join**(*path*, */*, **paths*)

聰明地連接一個或多個路徑段。回傳值是 *path* 和 **paths* 的所有成員的串聯，每個非空部分後面都有一個目🄵分隔符號，除了最後一個部分。🄵句話🄵，如果最後一個部分🄵空或以分隔符號結尾，結果只會以分隔符號結尾。如果一個段是🄵對路徑（在 Windows 上需要驅動機和根），則忽略所有之前的段，🄵從🄵對路徑段繼續連接。

在 Windows 上，當遇到根路徑段（例如，`r'\foo'`）時，驅動機不會被重置。如果一個段位於不同的驅動機上，或者是🄵對路徑，則將忽略所有之前的段🄵重置驅動機。請注意，由於每個驅動機都有目前目🄵，`os.path.join("c:", "foo")` 表示相對於驅動機 `C:` 的目前目🄵的路徑（即 `c:foo`），而不是 `c:\foo`。

在 3.6 版的變更: *path* 和 *paths* 接受*path-like object* 作🄵參數。

os.path.**normcase**(*path*, */*)

將路徑名的大小寫規範化。在 Windows 上，將路徑名中的所有字元轉🄵🄵小寫，🄵將正斜🄵轉🄵🄵反斜🄵。在其他作業系統上，回傳原始路徑。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**normpath**(*path*)

通過合🄵多余的分隔符號和上層引用來標准化路徑名，使得 `A//B`、`A/B/`、`A/./B` 和 `A/foo/../B` 都變成 `A/B`。這種字串操作可能會改變包含符號連結的路徑的含義。在 Windows 上，它將正斜🄵轉🄵🄵反斜🄵。要標准化大小寫，請使用*`normcase()`*。

> **備🄵**
>
> 在 POSIX 系統中，根據 IEEE Std 1003.1 2013 版; 4.13 Pathname Resolution 標准，如果一個路徑名恰好以兩個斜🄵開頭，則在前導字元後的第一個部分可能會以由實作品自行定義的方式解釋，雖然多於兩個前導字元應該被視🄵單個字元。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**realpath**(*path*, */*, *, *strict=False*)

回傳指定檔案名稱的規範路徑，會消除路徑中遇到的所有符號連結（如果作業系統支援）。在 Windows 上，此函式還會解析 MS-DOS（也稱🄵 8.3）樣式的名稱，例如 `C:\\PROGRA~1` 會被解析🄵 `C:\\Program Files`。

By default, the path is evaluated up to the first component that does not exist, is a symlink loop, or whose evaluation raises *`OSError`*. All such components are appended unchanged to the existing part of the path.

Some errors that are handled this way include "access denied", "not a directory", or "bad argument to internal function". Thus, the resulting path may be missing or inaccessible, may still contain links or loops, and may traverse non-directories.

This behavior can be modified by keyword arguments:

If *strict* is `True`, the first error encountered when evaluating the path is re-raised. In particular, `FileNotFoundError` is raised if *path* does not exist, or another `OSError` if it is otherwise inaccessible.

If *strict* is `os.path.ALLOW_MISSING`, errors other than `FileNotFoundError` are re-raised (as with `strict=True`). Thus, the returned path will not contain any symbolic links, but the named file and some of its parent directories may be missing.

備🄵

此函式模擬作業系統使路徑成🄴規範的過程，Windows 和 UNIX 之間在鏈接和後續路徑部份交互方式方面略有不同。

作業系統的 API 會根據需要自動使路徑正則，因此通常不需要呼叫此函式。

在 3.6 版的變更: 接受一個*path-like object*。

在 3.8 版的變更: 在 Windows 上，現在會解析符號連結和連接點。

在 3.10 版的變更: 新增 *strict* 參數。

在 3.14 版的變更: The `ALLOW_MISSING` value for the *strict* parameter was added.

os.path.**ALLOW_MISSING**

Special value used for the *strict* argument in `realpath()`.

在 3.14 版被加入.

os.path.**relpath**(*path*, *start=os.curdir*)

從目前目🄴或可選的 *start* 目🄴回傳到 *path* 的相對檔案路徑。這是一個路徑計算：不會存取檔案系統來確認 *path* 或 *start* 的存在或屬性。在 Windows 上，當 *path* 和 *start* 在不同的驅動機上時，會引發`ValueError`。

start 的預設值🄴`os.curdir`。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**samefile**(*path1*, *path2*, /)

如果兩個路徑名引數指向同一個檔案或目🄴，則回傳 `True`。這是通過設備編號和 i-node 編號來確定的，如果對任一路徑名的`os.stat()` 呼叫失敗，則會引發🄴常。

在 3.2 版的變更: 新增對 Windows 的支援。

在 3.4 版的變更: 現在在 Windows 上使用與其他所有平台相同的實作方式。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**sameopenfile**(*fp1*, *fp2*)

如果文件描述符 *fp1* 和 *fp2* 指向同一個檔案，則回傳 `True`。

在 3.2 版的變更: 新增對 Windows 的支援。

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**samestat**(*stat1*, *stat2*, /)

如果 stat 值組 *stat1* 和 *stat2* 指向同一個檔案，則回傳 `True`。這些結構可能由`os.fstat()`、`os.lstat()` 或`os.stat()` 回傳。此函式使用`samefile()` 和`sameopenfile()` 實現了底層比較。

在 3.4 版的變更: 新增對 Windows 的支援。

os.path.**split**(*path*, /)

將路徑名 *path* 拆分🄴 `(head, tail)` 一對，其中 *tail* 是最後一個路徑名部份，*head* 是在它之前的所有部分。*tail* 部分不會包含斜🄵；如果 *path* 以斜🄵結尾，則 *tail* 將🄴空。如果 *path* 中🄵有斜🄵，則 *head* 將🄴空。如果 *path* 🄴空，則 *head* 和 *tail* 都🄴空。除非 *head* 是根目🄵（僅有一個或多個斜🄵），

否則從 *head* 中🄴除尾部的斜🄵。在所有情🄵下，`join(head, tail)` 回傳指向與 *path* 相同位置的路徑（但字串可能不同）。還可以參考函式`dirname()` 和`basename()`。

在 3.6 版的變更: 接受一個*path-like object*。

`os.path.`**`splitdrive`**(*path*, */*)

將路徑名 *path* 拆分🄵 `(drive, tail)` 一對，其中 *drive* 是🄵載點或空字串。在不使用驅動機規範的系統上，*drive* 將始終🄵空字串。在所有情🄵下，`drive + tail` 將與 *path* 相同。

在 Windows 上，將路徑名拆分🄵驅動機或 UNC 共享點以及相對路徑。

如果路徑包含驅動機字母，則 *drive* 將包含從頭到冒號（包括冒號）的所有🄵容:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

如果路徑包含 UNC 路徑，則驅動機將包含主機名和共享名:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

在 3.6 版的變更: 接受一個*path-like object*。

`os.path.`**`splitroot`**(*path*, */*)

將路徑名 *path* 拆分🄵一個 3 項值組 `(drive, root, tail)`，其中 *drive* 是設備名稱或🄵載點，*root* 是驅動機後的分隔符字串，*tail* 是在根後的所有🄵容。這些項目中的任何一個都可能是空字串。在所有情🄵下，`drive + root + tail` 將與 *path* 相同。

在 POSIX 系統上，*drive* 始終🄵空。*root* 可能🄵空（如果 *path* 是相對路徑），一個斜🄵（如果 *path* 是🄵對路徑），或者兩個斜🄵（根據 IEEE Std 1003.1-2017; 4.13 Pathname Resolution 的實作定義）。例如:

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

在 Windows 上，*drive* 可能🄵空、驅動機名稱、UNC 共享或設備名稱。*root* 可能🄵空，斜🄵或反斜🄵。例如:

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

在 3.12 版被加入.

`os.path.`**`splitext`**(*path*, */*)

將路徑名 *path* 拆分🄵一對 `(root, ext)`，使得 `root + ext == path`，且副檔名 *ext* 🄵空或以點開頭且最多包含一個點 (period)。

如果路徑不包含副檔名，則 *ext* 將🄵 `''`:

```
>>> splitext('bar')
('bar', '')
```

如果路徑包含副檔名，則 *ext* 將設置🄵該副檔名，包括前導的點。請注意，前面的點將被忽略:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

路徑的最後一個部份的前導點被認🄴是根的一部分：

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

在 3.6 版的變更: 接受一個*path-like object*。

os.path.**supports_unicode_filenames**

如果可以使用任意的 Unicode 字串作🄵檔案名（在檔案系統所施加的限制範圍🄴），則回傳 `True`。

11.3 stat --- 直譯 stat() 的結果

原始碼： Lib/stat.py

The *stat* module defines constants and functions for interpreting the results of *os.stat()*, *os.fstat()* and *os.lstat()* (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

在 3.4 版的變更: The stat module is backed by a C implementation.

stat 模組定義了以下函式，用於測試特定檔案類型：

stat.**S_ISDIR**(*mode*)

Return non-zero if the mode is from a directory.

stat.**S_ISCHR**(*mode*)

Return non-zero if the mode is from a character special device file.

stat.**S_ISBLK**(*mode*)

Return non-zero if the mode is from a block special device file.

stat.**S_ISREG**(*mode*)

Return non-zero if the mode is from a regular file.

stat.**S_ISFIFO**(*mode*)

Return non-zero if the mode is from a FIFO (named pipe).

stat.**S_ISLNK**(*mode*)

Return non-zero if the mode is from a symbolic link.

stat.**S_ISSOCK**(*mode*)

Return non-zero if the mode is from a socket.

stat.**S_ISDOOR**(*mode*)

Return non-zero if the mode is from a door.

在 3.4 版被加入.

stat.**S_ISPORT**(*mode*)

Return non-zero if the mode is from an event port.

在 3.4 版被加入.

stat.**S_ISWHT**(*mode*)

Return non-zero if the mode is from a whiteout.

在 3.4 版被加入.

Two additional functions are defined for more general manipulation of the file's mode:

stat.**S_IMODE**(*mode*)

Return the portion of the file's mode that can be set by *os.chmod()*---that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.**S_IFMT**(*mode*)

Return the portion of the file's mode that describes the file type (used by the `S_IS*()` functions above).

Normally, you would use the `os.path.is*()` functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn't handled by *os.path*, like the tests for block and character devices.

範例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a human readable string:

stat.**filemode**(*mode*)

Convert a file's mode to a string of the form '-rwxrwxrwx'.

在 3.3 版被加入.

在 3.4 版的變更: The function supports *S_IFDOOR*, *S_IFPORT* and *S_IFWHT*.

All the variables below are simply symbolic indexes into the 10-tuple returned by *os.stat()*, *os.fstat()* or *os.lstat()*.

stat.**ST_MODE**

Inode protection mode.

stat.**ST_INO**

Inode number.

stat.**ST_DEV**

Device inode resides on.

stat.**ST_NLINK**

Number of links to the inode.

stat.**ST_UID**

User id of the owner.

stat.**ST_GID**

Group id of the owner.

stat.**ST_SIZE**

Size in bytes of a plain file; amount of data waiting on some special files.

stat.**ST_ATIME**

Time of last access.

stat.**ST_MTIME**

Time of last modification.

stat.**ST_CTIME**

The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to *os.stat()*, *os.fstat()*, or *os.lstat()*; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the *ST_MODE* field.

Use of the functions above is more portable than use of the first set of flags:

stat.**S_IFSOCK**

Socket.

stat.**S_IFLNK**

Symbolic link.

stat.**S_IFREG**

Regular file.

stat.**S_IFBLK**

Block device.

stat.**S_IFDIR**

Directory.

stat.**S_IFCHR**

Character device.

stat.**S_IFIFO**

FIFO.

stat.**S_IFDOOR**

Door.

在 3.4 版被加入.

stat.**S_IFPORT**

Event port.

在 3.4 版被加入.

stat.**S_IFWHT**

Whiteout.

在 3.4 版被加入.

備🄵

S_IFDOOR, *S_IFPORT* or *S_IFWHT* are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of *os.chmod()*:

stat.**S_ISUID**
Set UID bit.

stat.**S_ISGID**
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the *S_ISGID* bit set. For a file that does not have the group execution bit (*S_IXGRP*) set, the set-group-ID bit indicates mandatory file/record locking (see also *S_ENFMT*).

stat.**S_ISVTX**
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.**S_IRWXU**
Mask for file owner permissions.

stat.**S_IRUSR**
Owner has read permission.

stat.**S_IWUSR**
Owner has write permission.

stat.**S_IXUSR**
Owner has execute permission.

stat.**S_IRWXG**
Mask for group permissions.

stat.**S_IRGRP**
Group has read permission.

stat.**S_IWGRP**
Group has write permission.

stat.**S_IXGRP**
Group has execute permission.

stat.**S_IRWXO**
Mask for permissions for others (not in group).

stat.**S_IROTH**
Others have read permission.

stat.**S_IWOTH**
Others have write permission.

stat.**S_IXOTH**
Others have execute permission.

stat.**S_ENFMT**
System V file locking enforcement. This flag is shared with *S_ISGID*: file/record locking is enforced on files that do not have the group execution bit (*S_IXGRP*) set.

stat.**S_IREAD**
Unix V7 synonym for *S_IRUSR*.

stat.**S_IWRITE**
Unix V7 synonym for *S_IWUSR*.

stat.**S_IEXEC**
Unix V7 synonym for *S_IXUSR*.

The following flags can be used in the *flags* argument of *os.chflags()*:

stat.**UF_SETTABLE**
All user settable flags.

在 3.13 版被加入.

stat.**UF_NODUMP**
Do not dump the file.

stat.**UF_IMMUTABLE**
The file may not be changed.

stat.**UF_APPEND**
The file may only be appended to.

stat.**UF_OPAQUE**
The directory is opaque when viewed through a union stack.

stat.**UF_NOUNLINK**
The file may not be renamed or deleted.

stat.**UF_COMPRESSED**
The file is stored compressed (macOS 10.6+).

stat.**UF_TRACKED**
Used for handling document IDs (macOS)

在 3.13 版被加入.

stat.**UF_DATAVAULT**
The file needs an entitlement for reading or writing (macOS 10.13+)

在 3.13 版被加入.

stat.**UF_HIDDEN**
The file should not be displayed in a GUI (macOS 10.5+).

stat.**SF_SETTABLE**
All super-user changeable flags

在 3.13 版被加入.

stat.**SF_SUPPORTED**
All super-user supported flags

可用性: macOS

在 3.13 版被加入.

stat.**SF_SYNTHETIC**
All super-user read-only synthetic flags

可用性: macOS

在 3.13 版被加入.

stat.**SF_ARCHIVED**
The file may be archived.

stat.**SF_IMMUTABLE**
The file may not be changed.

stat.**SF_APPEND**
The file may only be appended to.

stat.**SF_RESTRICTED**
The file needs an entitlement to write to (macOS 10.13+)

在 3.13 版被加入.

stat.**SF_NOUNLINK**
The file may not be renamed or deleted.

stat.**SF_SNAPSHOT**
The file is a snapshot file.

stat.**SF_FIRMLINK**
The file is a firmlink (macOS 10.15+)

在 3.13 版被加入.

stat.**SF_DATALESS**
The file is a dataless object (macOS 10.15+)

在 3.13 版被加入.

See the *BSD or macOS systems man page *chflags(2)* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the st_file_attributes member returned by *os.stat()*. See the Windows API documentation for more detail on the meaning of these constants.

stat.**FILE_ATTRIBUTE_ARCHIVE**
stat.**FILE_ATTRIBUTE_COMPRESSED**
stat.**FILE_ATTRIBUTE_DEVICE**
stat.**FILE_ATTRIBUTE_DIRECTORY**
stat.**FILE_ATTRIBUTE_ENCRYPTED**
stat.**FILE_ATTRIBUTE_HIDDEN**
stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**
stat.**FILE_ATTRIBUTE_NORMAL**
stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**
stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**
stat.**FILE_ATTRIBUTE_OFFLINE**
stat.**FILE_ATTRIBUTE_READONLY**
stat.**FILE_ATTRIBUTE_REPARSE_POINT**
stat.**FILE_ATTRIBUTE_SPARSE_FILE**
stat.**FILE_ATTRIBUTE_SYSTEM**
stat.**FILE_ATTRIBUTE_TEMPORARY**
stat.**FILE_ATTRIBUTE_VIRTUAL**
在 3.5 版被加入.

On Windows, the following constants are available for comparing against the st_reparse_tag member returned by *os.lstat()*. These are well-known constants, but are not an exhaustive list.

stat.**IO_REPARSE_TAG_SYMLINK**
stat.**IO_REPARSE_TAG_MOUNT_POINT**

stat.**IO_REPARSE_TAG_APPEXECLINK**

在 3.8 版被加入.

11.4 filecmp --- 檔案與目🄵比較

原始碼： Lib/filecmp.py

The *filecmp* module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the *difflib* module.

filecmp 模組定義了以下函式：

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)

Compare the files named *f1* and *f2*, returning `True` if they seem equal, `False` otherwise.

If *shallow* is true and the *os.stat()* signatures (file type, size, and modification time) of both files are identical, the files are taken to be equal.

Otherwise, the files are treated as different if their sizes or contents differ.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the *os.stat()* information for the file changes. The entire cache may be cleared using *clear_cache()*.

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)

Compare the files in the two directories *dir1* and *dir2* whose names are given by *common*.

Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don't, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don't exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `'d/e'` will each be in one of the three returned lists.

filecmp.**clear_cache**()

Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

在 3.4 版被加入.

11.4.1 The dircmp class

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*, ***, *shallow=True*)

Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to *filecmp.DEFAULT_IGNORES*. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The *dircmp* class compares files by doing *shallow* comparisons as described for *filecmp.cmp()* by default using the *shallow* parameter.

在 3.13 版的變更: Added the *shallow* parameter.

The *dircmp* class provides the following methods:

report()

Print (to *sys.stdout*) a comparison between *a* and *b*.

report_partial_closure()

Print a comparison between *a* and *b* and common immediate subdirectories.

report_full_closure()

Print a comparison between *a* and *b* and common subdirectories (recursively).

The *dircmp* class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left

The directory *a*.

right

The directory *b*.

left_list

Files and subdirectories in *a*, filtered by *hide* and *ignore*.

right_list

Files and subdirectories in *b*, filtered by *hide* and *ignore*.

common

Files and subdirectories in both *a* and *b*.

left_only

Files and subdirectories only in *a*.

right_only

Files and subdirectories only in *b*.

common_dirs

Subdirectories in both *a* and *b*.

common_files

Files in both *a* and *b*.

common_funny

Names in both *a* and *b*, such that the type differs between the directories, or names for which *os.stat()* reports an error.

same_files

Files which are identical in both *a* and *b*, using the class's file comparison operator.

diff_files

Files which are in both *a* and *b*, whose contents differ according to the class's file comparison operator.

funny_files

Files which are in both *a* and *b*, but could not be compared.

subdirs

A dictionary mapping names in *common_dirs* to *dircmp* instances (or MyDirCmp instances if this instance is of type MyDirCmp, a subclass of *dircmp*).

在 3.10 版的變更: Previously entries were always *dircmp* instances. Now entries are the same type as *self*, if *self* is a subclass of *dircmp*.

filecmp.**DEFAULT_IGNORES**

在 3.4 版被加入.

List of directories ignored by *dircmp* by default.

Here is a simplified example of using the `subdirs` attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.5 `tempfile` --- 生成臨時檔案和目🄵

原始碼： Lib/tempfile.py

該 module（模組）用於建立臨時檔案和目🄵，它可以在所有有支援的平臺上使用。*TemporaryFile*、*NamedTemporaryFile*、*TemporaryDirectory* 和*SpooledTemporaryFile* 是有自動清除功能的高階介面，可作🄵情境管理器 *(context manager)* 使用。*mkstemp()* 和*mkdtemp()* 是低階函式，使用完畢後需手動清理。

所有可被使用者呼叫的函式和建構函式都帶有可以設定臨時檔案和臨時目🄵的路徑和名稱的引數。此 module 所使用的檔名🄵一個隨機字元組成的字串，這讓檔案可以更安全地在共享的臨時目🄵中被建立。🄵了維持向後相容性，引數的順序會稍微奇怪，所以🄵了讓程式更容易被理解，建議使用關鍵字引數。

這個 module 定義了以下可供使用者呼叫的項目：

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

回傳一個可當作臨時儲存區域的類檔案物件。建立該檔案使用了與*mkstemp()* 相同的安全規則。它將在關閉（包括當物件被垃圾回收 (garbage collect) 時的隱式關閉）後立即銷🄴。在 Unix 下，該檔案所在的目🄵可能根本不被建立、或者在建立檔案後立即就被🄵除，其他平臺不支援此功能；你的程式不應依賴使用此功能建立的臨時檔案名稱，因🄵它在檔案系統中的名稱有可能是不可見的。

生成的物件可以作🄴情境管理器使用（參見範例）。完成情境或銷🄴臨時檔案物件後，臨時檔案將從檔案系統中🄵除。

mode 參數預設🄵 `'w+b'`，所以建立的檔案不用關閉就可以讀取或寫入。因🄵用的是二進位制模式，所以無論存的是什🄴資料，它在所有平臺上的行🄵都一致。*buffering*、*encoding*、*errors* 和 *newline* 的含義與*open()* 中的相同。

參數 *dir*、*prefix* 和 *suffix* 的含義和預設值都與它們在*mkstemp()* 中的相同。

在 POSIX 平臺上，回傳物件是真實的檔案物件。在其他平臺上，它是一個 file-like object，它的 `file` 屬性🄵底層的真實檔案物件。

如果可用且可運作，則使用*os.O_TMPFILE* 旗標（僅限於 Linux，需要 3.11 版本以上的核心）。

在不是 Posix 或 Cygwin 的平臺上，TemporaryFile 是 NamedTemporaryFile 的🄵名。

引發一個附帶引數 `fullpath` 的 `tempfile.mkstemp` 稽核事件。

在 3.5 版的變更: 如果可用，自此開始使用*os.O_TMPFILE* 旗標。

在 3.8 版的變更: 新增 *errors* 參數。

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*, *delete_on_close=True*)

此函式的操作與*TemporaryFile()* 完全相同，但存在以下差🄵：

- 此函式回傳一個保證在檔案系統中具有可見名稱的檔案。
- 🄵了管理指定檔案，它使用 *delete* 和 *delete_on_close* 參數擴充*TemporaryFile()* 來指定是否以及如何自動🄵除指定檔案。

回傳的物件始終是一個類檔案物件，其 `file` 屬性是底層的真實檔案物件。這個類檔案物件可以在 `with` 陳述式中使用，就像普通檔案一樣。臨時檔案的名稱可以從回傳的類檔案物件的 `name` 屬性中取得。在 Unix 上則與*`TemporaryFile()`* 不同，目🄵條目不會在檔案建立後立即被取消鏈接 (unlink)。

如果 *delete* 🄵 true（預設值）🄵且 *delete_on_close* 🄵 true（預設值），則檔案在關閉後會立即被🄵除。如果 *delete* 🄵 true 🄵且 *delete_on_close* 🄵 false，則僅在情境管理器退出時🄵除檔案，或者在類檔案物件完結時🄵除檔案。在這種情🄴下，🄵不總是保證能成功🄵除（請參🄵 `object.__del__()`）。如果 *delete* 🄵 false，則會忽略 *delete_on_close* 的值。

因此，要在關閉檔案後使用臨時檔案的名稱重新打開檔案，請確保在關閉時不🄵除檔案（將 *delete* 參數設定🄵 false），或者如果臨時檔案是以 `with` 陳述式建立，要將 *delete_on_close* 參數設定🄵 false。建議使用後者，因🄵它有助於在情境管理器退出時自動清理臨時檔案。

在臨時檔案仍處於打開狀態時再次按其名稱打開它，其作用方式如下：

- 在 POSIX 上，檔案始終可以再次打開。
- 在 Windows 上，確保至少滿足以下條件之一：
  - *delete* 🄵 false
  - 額外的 open 會共享🄵除存取權限（例如，通過使用旗標 `O_TEMPORARY` 來呼叫*`os.open()`*）
  - *delete* 🄵 true 但 *delete_on_close* 🄵 false。請注意，在這種情🄴下不共享🄵除存取權限的其他 open（例如透過🄴建的*`open()`* 建立）必須在退出情境管理器之前關閉，否則情境管理器上的*`os.unlink()`* 呼叫退出將失敗🄵出現*`PermissionError`*。

在 Windows 上，如果 *delete_on_close* 🄵 false，🄵且檔案是在使用者缺乏🄵除存取權限的目🄵中建立的，則情境管理器退出時的*`os.unlink()`* 呼叫將失敗，🄵引發*`PermissionError`*。當 *delete_on_close* 🄵 true 時，不會發生這種情🄵，因🄵🄵除存取權限是由 open 來要求的，如果未授予存取權限則會立即失敗。

（僅）在 POSIX 上，因使用 SIGKILL 而被終止的行程無法自動🄵除它建立的任何 NamedTemporaryFiles。

引發一個附帶引數 `fullpath` 的 `tempfile.mkstemp` 稽核事件。

在 3.8 版的變更: 新增 *errors* 參數。

在 3.12 版的變更: 新增 *delete_on_close* 參數。

class tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

此類🄵執行的操作與*`TemporaryFile()`* 完全相同，但會將資料排存 (spool) 於在記憶體中，直到檔案大小超過 *max_size*，或檔案的*`fileno()`* 方法被呼叫🄵止，此時資料會被寫入磁碟，🄵且之後的操作與*`TemporaryFile()`* 相同。

rollover()

生成的檔案物件有一個額外的方法 `rollover()`，忽略檔案大小🄵立即將其寫入磁碟。

回傳的物件是 file-like object，它的 `_file` 屬性是*`io.BytesIO`* 或*`io.TextIOWrapper`* 物件（取🄵於指定的是二進位制模式還是文字模式）或真實的檔案物件（取🄴於是否已呼叫*`rollover()`*）。file-like object 可以像普通檔案一樣在 `with` 陳述式中使用。

在 3.3 版的變更: 現在，檔案的截斷方法 (truncate method) 可接受一個 *size* 引數。

在 3.8 版的變更: 新增 *errors* 參數。

在 3.11 版的變更: 完全實作*`io.BufferedIOBase`* 和*`io.TextIOBase`* 抽象基底類🄵（取🄵於指定的是二進位還是文本 *mode*）。

class tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*, ***, *delete=True*)

此類🄵會使用與*`mkdtemp()`* 相同安全規則來建立一個臨時目🄵。回傳物件可當作情境管理器使用

(參見範例)。在完成情境或銷🄵臨時目🄵物件時，新建立的臨時目🄵及其所有🄵容會從檔案系統中被移除。

name

可以從回傳物件的 `name` 屬性中找到的臨時目🄵名稱。當回傳的物件用作情境管理器時，這個 `name` 會作🄵 `with` 陳述句中 `as` 子句的目標（如果有 as 的話）。

cleanup()

此目🄵可透過呼叫 `cleanup()` 方法來顯式地清理。如果 *ignore_cleanup_errors* 🄵 true，則在顯式或隱式清理期間出現的未處理例外（例如在 Windows 上移除開🄵的檔案而引發的*PermissionError*）將被忽略，🄵且剩余的可移除條目會「🄵可能」地被🄵除。在其他情🄵下，錯誤將在任何情境清理發生時被引發（`cleanup()` 呼叫、退出情境管理器、物件被作🄵垃圾回收或直譯器關閉等）。

delete 參數可以停用在退出情境時對目🄵樹的清理，雖然停用情境管理器在退出情境時所🄵取的操作似乎不常見，但它在除錯期間或當你需要基於其他邏輯的清理行🄵時會非常有用。

引發一個附帶引數 `fullpath` 的 `tempfile.mkdtemp` 稽核事件。

在 3.2 版被加入.

在 3.10 版的變更: 新增 *ignore_cleanup_errors* 參數。

在 3.12 版的變更: 新增 *delete* 參數。

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

盡可能以最安全的方式建立一個臨時檔案。假設所在平臺正確實作了*os.open()* 的*os.O_EXCL* 旗標，則建立檔案時不會有 race condition（競🄵條件）的情🄵。該檔案只能由建立者讀寫，如果所在平臺用 permission bit（許可權位元）來表示檔案是否可執行，則🄵有人有執行權。檔案描述器不會被子行程繼承。

與*TemporaryFile()* 不同，*mkstemp()* 使用者用完臨時檔案後需要自行將其🄵除。

如果 *suffix* 不是 `None` 則檔名將以該後綴結尾，若🄵 `None` 則🄵有後綴。*mkstemp()* 不會在檔名和後綴之間加點 (dot)，如果需要加一個點號，請將其放在 *suffix* 的開頭。

如果 *prefix* 不是 `None` 則檔名將以該字首開頭，若🄵 `None` 則使用預設前綴。預設前綴是*gettempprefix()* 或*gettempprefixb()* 函式的回傳值（自動呼叫合適的函式）。

如果 *dir* 不🄵 `None` 則在指定的目🄵建立檔案，若🄵 `None` 則使用預設目🄵。預設目🄵是從一個相依於平臺的列表中選擇出來的，但是使用者可以設定 *TMPDIR*、*TEMP* 或 *TMP* 環境變數來設定目🄵的位置。因此，不能保證生成的臨時檔案路徑是使用者友善的，比如透過 `os.popen()` 將路徑傳遞給外部命令時仍需要加引號 (quoting)。

如果 *suffix*、*prefix* 和 *dir* 中的任何一個不是 `None`，就要保證它們資料型🄵相同。如果它們是位元組串，則回傳名稱的型🄵就是位元組串而非字串。如果不想遵循預設行🄵但又想要回傳值是位元組串型🄵，請傳入 `suffix=b''`。

如果指定了 *text* 且🄵真值，檔案會以文字模式開🄵。否則，檔案（預設）以二進位制模式開🄵。

mkstemp() 回傳一個 tuple，tuple 中，第一個元素是一個作業系統層級 (OS-level) 控制代碼，指向一個開🄵的檔案（如同*os.open()* 的回傳值），第二元素是該檔案的🄵對路徑。

引發一個附帶引數 `fullpath` 的 `tempfile.mkstemp` 稽核事件。

在 3.5 版的變更: 現在，*suffix*、*prefix* 和 *dir* 可以以位元組串型🄵按順序提供，以獲得位元組串型🄵的回傳值。在之前只允許使用字串。*suffix* 和 *prefix* 現在可以接受 `None`，🄵且預設🄵 `None` 以使用合適的預設值。

在 3.6 版的變更: *dir* 參數現在可接受一個類路徑物件 (*path-like object*)。

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)

盡可能以最安全的方式建立一個臨時目🄵，建立該目🄵時不會有 race condition 的情🄵，該目🄵只能由建立者讀取、寫入和搜尋。

mkdtemp() 的使用者用完臨時目🄵後需要自行將其🄵除。

引數 *prefix*、*suffix* 和 *dir* 的含義與它們在*mkstemp()* 中相同。

mkdtemp() 回傳新目🄴的🄵對路徑名稱。

引發一個附帶引數 `fullpath` 的 `tempfile.mkdtemp` 稽核事件。

在 3.5 版的變更: 現在，*suffix*、*prefix* 和 *dir* 可以以位元組串型🄴按順序提供，以獲得位元組串型🄴的回傳值。在之前只允許使用字串。*suffix* 和 *prefix* 現在可以接受 `None`，🄴且預設🄴 `None` 以使用合適的預設值。

在 3.6 版的變更: *dir* 參數現在可接受一個類路徑物件 (*path-like object*)。

在 3.12 版的變更: *mkdtemp()* 現在都會回傳🄵對路徑，即使 *dir* 是相對路徑。

`tempfile.`**`gettempdir`**`()`

回傳儲存臨時檔案的目🄵名稱。這設定了此 module 所有函式 *dir* 引數的預設值。

Python 搜尋標准目🄵列表來找到呼叫者可以在其中建立檔案的目🄵。這個列表是:

1. `TMPDIR` 環境變數指向的目🄵。
2. `TEMP` 環境變數指向的目🄵。
3. `TMP` 環境變數指向的目🄵。
4. 與平臺相關的位置:
   - 在 Windows 上，目🄵依次🄵 `C:\TEMP`、`C:\TMP`、`\TEMP` 和 `\TMP`。
   - 在所有其他平臺上，目🄵依次🄵 `/tmp`、`/var/tmp` 和 `/usr/tmp`。
5. 不得已時，使用目前工作目🄵。

搜尋的結果會被 cache（快取）起來，請見下面*tempdir* 的描述。

在 3.10 版的變更: 回傳一個字串。在之前的版本中它會回傳任意*tempdir* 的值而不考慮它的型🄵，只要它不🄵 `None`。

`tempfile.`**`gettempdirb`**`()`

與*gettempdir()* 相同，但回傳值🄵位元組串型🄵。

在 3.5 版被加入.

`tempfile.`**`gettempprefix`**`()`

回傳用於建立臨時檔案的檔名前綴，它不包含目🄵部分。

`tempfile.`**`gettempprefixb`**`()`

與*gettempprefix()* 相同，但回傳值🄵位元組串型🄵。

在 3.5 版被加入.

此 module 使用一個全域性變數來儲存由*gettempdir()* 回傳的臨時檔案使用的目🄵路徑。它可被直接設定以覆蓋選擇過程，但不建議這樣做。此 module 中的所有函式都接受一個 *dir* 引數，它可被用於指定目🄵。這是個推薦的做法，它不會透過改變全域性 API 行🄵而對其他不預期此行🄵的程式造成影響。

`tempfile.`**`tempdir`**

當被設🄵 `None` 以外的值時，此變數會🄵此 module 所定義函式的引數 *dir* 定義預設值，包括確定其型🄵🄵位元組串還是字串。它不可以🄵*path-like object*。

如果在呼叫除*gettempprefix()* 外的上述任何函式時 `tempdir` 🄵 `None` (預設值) 則它會按照*gettempdir()* 中所描述的演算法來初始化。

> **備🄵**
>
> 請注意如果你將 `tempdir` 設🄵位元組串值，會有一個麻煩的副作用：*mkstemp()* 和*mkdtemp()* 的全域性預設回傳型🄵會在🄵有提供明顯字串型🄵的 `prefix`、`suffix` 或 `dir` 時被改🄵位元組串。請不要編寫預期此行🄵或依賴於此行🄵的程式。這個奇怪的行🄵是🄵了維持與以往實作版本的相容性。

11.5.1 范例

以下是*tempfile* module 的一些常見用法範例：

```
>>> import tempfile

# 建立一個臨時檔案🄵寫入一些資料
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# 關閉檔案，檔案會被🄵除
>>> fp.close()

# 使用情境管理器建立臨時檔案
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# 檔案現在已關閉🄵🄵除

# 使用情境管理器建立臨時檔案
# 關閉檔案，使用名稱再次開🄵檔案
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as fp:
...     fp.write(b'Hello world!')
...     fp.close()
... # 檔案已被關閉，但未被🄵除
... # 透過其名稱再次開🄵檔案
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# 現在檔案已被🄵除

# 使用情境管理器建立臨時目🄵
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# 目🄵及其🄵容已被🄵除
```

11.5.2 已🄵用的函式和變數

在過去，建立臨時檔案首先使用*mktemp()* 函式生成一個檔名，然後使用該檔名建立檔案。不幸的是這🄵不安全的，因🄵在呼叫*mktemp()* 與隨後嘗試建立檔案之間的時間🄴，其他程式可能會使用該名稱建立檔案。解🄵方案是將兩個步驟結合起來，🄵立即建立檔案。這個方案目前被*mkstemp()* 和上述其他函式所🄵用。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

在 2.3 版之後被🄵用: 使用*mkstemp()* 代替。

回傳一個在呼叫本方法時不存在檔案的🄵對路徑。引數 *prefix*、*suffix* 和 *dir* 與*mkstemp()* 中所用的類似，除了在於不支援位元組串型🄵的檔名且不支援 `suffix=None` 和 `prefix=None`。

> **⚠ 警告**
>
> 使用此功能可能會在程式中引入安全漏洞。當你開始使用本方法回傳的檔案執行任何操作時，可能有人已經捷足先登了。*mktemp()* 的功能可以很輕松地用帶有 `delete=False` 參數

的*NamedTemporaryFile()* 代替：

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.6 `glob` --- Unix 風格的路徑名稱模式擴展

原始碼： Lib/glob.py

---

glob 模組根據 Unix shell 使用的規則查找與指定模式匹配的所有路徑名稱，結果以任意順序回傳。🄵有波浪號 (tilde) 擴展，但是 `*`、`?` 和用 `[]` 表示的字元範圍將被正確匹配。這是透過同時使用*os.scandir()* 和*fnmatch.fnmatch()* 函式來完成的，而🄵有實際呼叫 subshell。

請注意，以點 (`.`) 開頭的檔案只能與同樣以點開頭的模式匹配，這與*fnmatch.fnmatch()* 或*pathlib.Path.glob()* 不同。（對於波浪號和 shell 變數擴展，請使用*os.path.expanduser()* 和*os.path.expandvars()*。）

對於文本 (literal) 匹配，將元字元 (meta-character) 括在方括號中。例如，`'[?]'` 會匹配 `'?'` 字元。

glob 模組定義了以下函式：

glob.**glob**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

回傳與 *pathname* 匹配、可能🄵空的路徑名稱 list，它必須是包含路徑規範的字串。*pathname* 可以是🄵對的（如 `/usr/src/Python-1.5/Makefile`）或相對的（如 `../../Tools/*/*.gif`），🄵且可以包含 shell 樣式的通用字元 (wildcard)。已損壞的符號連接也會（如同在 shell）被包含在結果中。結果是否排序取🄵於檔案系統 (file system)。如果在呼叫此函式期間🄵除或新增滿足條件的檔案，則結果不一定會包含該檔案的路徑名稱。

如果 *root_dir* 不是 `None`，它應該是一個指定搜尋根目🄵的*path-like object*。它在呼叫它之前更改目前目🄵的影響與*glob()* 相同。如果 *pathname* 是相對的，結果將包含相對於 *root_dir* 的路徑。

此函式可以支援以 *dir_fd* 參數使用相對目🄵描述器的路徑。

如果 *recursive* 🄵真，模式”`**`” 將匹配任何檔案、零個或多個目🄵、子目🄵和目🄵的符號連結。如果模式後面有*os.sep* 或*os.altsep* 那🄵檔案將不會被匹配。

如果 *include_hidden* 🄵真，”`**`” 模式將匹配被隱藏的目🄵。

引發一個附帶引數 `pathname`、`recursive` 的稽核事件 `glob.glob`。

引發一個附帶引數 `pathname`、`recursive`、`root_dir`、`dir_fd` 的稽核事件 `glob.glob/2`。

> **備🄵**
>
> 在大型目🄵樹中使用”`**`” 模式可能會消耗過多的時間。

> **備🄵**
>
> This function may return duplicate path names if *pathname* contains multiple ”`**`” patterns and *recursive* is true.

在 3.5 版的變更: 支援以"`**`" 使用遞🄵 glob。

在 3.10 版的變更: 新增 *root_dir* 與 *dir_fd* 參數。

在 3.11 版的變更: 新增 *include_hidden* 參數。

`glob.`**`iglob`**(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

回傳一個會🄵生與`glob()` 相同的值的*iterator* ， 而不是同時儲存全部的值。

引發一個附帶引數 `pathname`、`recursive` 的稽核事件 `glob.glob`。

引發一個附帶引數 `pathname`、`recursive`、`root_dir`、`dir_fd` 的稽核事件 `glob.glob/2`。

備🄵

This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

在 3.5 版的變更: 支援以"`**`" 使用遞🄵 glob。

在 3.10 版的變更: 新增 *root_dir* 與 *dir_fd* 參數。

在 3.11 版的變更: 新增 *include_hidden* 參數。

`glob.`**`escape`**(*pathname*)

跳🄵 (escape) 所有特殊字元（`'?'`、`'*'` 和 `'['`）。如果你想匹配其中可能包含特殊字元的任意文本字串，這將會很有用。驅動器 (drive)/UNC 共享點 (sharepoints) 中的特殊字元不會被跳🄵，例如在 Windows 上，`escape('//?/c:/Quo vadis?.txt')` 會回傳 `'//?/c:/Quo vadis[?].txt'`。

在 3.4 版被加入.

`glob.`**`translate`**(*pathname*, *, *recursive=False*, *include_hidden=False*, *seps=None*)

Convert the given path specification to a regular expression for use with `re.match()`. The path specification can contain shell-style wildcards.

舉例來🄵:

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True, include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

Path separators and segments are meaningful to this function, unlike `fnmatch.translate()`. By default wildcards do not match path separators, and `*` pattern segments match precisely one path segment.

If *recursive* is true, the pattern segment "`**`" will match any number of path segments.

If *include_hidden* is true, wildcards can match path segments that start with a dot (`.`).

A sequence of path separators may be supplied to the *seps* argument. If not given, `os.sep` and `altsep` (if available) are used.

也參考

`pathlib.PurePath.full_match()` and `pathlib.Path.glob()` methods, which call this function to implement pattern matching and globbing.

在 3.13 版被加入.

11.6.1 范例

例如，在一個包含以下檔案的目🄵：`1.gif`、`2.txt`、`card.gif`，和一個僅包含 `3.txt` 檔案的子目🄵 `sub`，*`glob()`* 將🄵生以下結果。請注意路徑的任何前導部分是如何保留的。

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

如果目🄵包含以 `.` 開頭的檔案，則預設情🄵下不會去匹配到它們。例如，一個包含 `card.gif` 和 `.card.gif` 的目🄵：

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

也參考

`fnmatch` 模組提供了 shell 風格檔案名（不是路徑）的擴展

也參考

`pathlib` 模組提供高階路徑物件。

11.7 `fnmatch` --- Unix 檔案名稱模式比對

原始碼： Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are *not* the same as regular expressions (which are documented in the *`re`* module). The special characters used in shell-style wildcards are:

Pattern	含義
`*`	matches everything
`?`	matches any single character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

Note that the filename separator (`'/'` on Unix) is *not* special to this module. See module *`glob`* for pathname expansion (*`glob`* uses *`filter()`* to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

Unless stated otherwise, "filename string" and "pattern string" either refer to *str* or ISO-8859-1 encoded *bytes* objects. Note that the functions documented below do not allow to mix a bytes pattern with a str filename, and vice-versa.

Finally, note that *functools.lru_cache()* with a *maxsize* of 32768 is used to cache the (typed) compiled regex patterns in the following functions: *fnmatch()*, *fnmatchcase()*, *filter()*, *filterfalse()*.

fnmatch.**fnmatch**(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning True or False. Both parameters are case-normalized using *os.path.normcase()*. *fnmatchcase()* can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

This example will print all file names in the current directory with the extension .txt:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning True or False; the comparison is case-sensitive and does not apply *os.path.normcase()*.

fnmatch.**filter**(*names*, *pat*)

Construct a list from those elements of the *iterable* of filename strings *names* that match the pattern string *pat*. It is the same as [n for n in names if fnmatch(n, pat)], but implemented more efficiently.

fnmatch.**filterfalse**(*names*, *pat*)

Construct a list from those elements of the *iterable* of filename strings *names* that do not match the pattern string *pat*. It is the same as [n for n in names if not fnmatch(n, pat)], but implemented more efficiently.

在 3.14 版被加入.

fnmatch.**translate**(*pat*)

Return the shell-style pattern *pat* converted to a regular expression for using with *re.match()*. The pattern is expected to be a *str*.

範例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

也參考

glob 模組
Unix shell-style path expansion.

11.8 linecache --- 隨機存取文字列

原始碼： Lib/linecache.py

---

linecache 模組允許從 Python 原始碼檔案中取得任何一行。當嘗試在程式🄵部進行最佳化，會使用快取，這是從單一檔案讀取許多行的常見情🄵。這被*traceback* 模組用來🄵取來自原始碼檔案的行，以包含在格式化的 traceback 中。

tokenize.open() 函式用來開🄵檔案。這個函式使用*tokenize.detect_encoding()* 來取得檔案的編碼；在🄵有編碼標記的情🄵下，檔案編碼預設🄵 UTF-8。

linecache 模組定義了下列函式：

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)

從名🄵 *filename* 的檔案中取得 *lineno* 行。這個函式不會🄵生例外 --- 它會在出錯時回傳 `''`（找到的行會包含終止🄵行字元）。

If *filename* indicates a frozen module (starting with `'<frozen '`), the function will attepmt to get the real file name from `module_globals['__file__']` if *module_globals* is not `None`.

如果找不到名🄵 *filename* 的檔案，函式會先檢查 *module_globals* 中是否有 **PEP 302** *__loader__* 載入器。如果有這樣一個載入器，而且它定義了一個 `get_source` 方法，這之後它就會🄵定原始碼行（如果 `get_source()` 回傳 `None`，那🄵就會回傳 `''`）。最後，如果 *filename* 是相對的檔案名稱，則會相對於模組搜尋路徑 `sys.path` 中的項目進行搜尋。

在 3.14 版的變更: Support *filename* of frozen modules.

linecache.**clearcache**()

清除快取。如果你不再需要先前使用*getline()* 讀取的檔案行數，請使用此函式。

linecache.**checkcache**(*filename=None*)

檢查快取是否有效。如果快取中的檔案可能已在磁碟上變更，而你需要更新的版本，請使用此功能。如果省略 *filename*，則會檢查快取中的所有項目。

linecache.**lazycache**(*filename*, *module_globals*)

即使 *module_globals* 在稍後的呼叫中是 `None`，也可以🄵取非檔案型模組的足🄵細節，以允許稍後透過*getline()* 取得其行。這可以避免在真正需要某一行之前進行 I/O 操作，而不必無限期地帶著模組的全域變數。

在 3.5 版被加入.

範例：

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.9 shutil —高階檔案操作

原始碼： Lib/shutil.py

The *shutil* module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the *os* module.

> **⚠ 警告**
>
> Even the higher-level file copying functions (*shutil.copy()*, *shutil.copy2()*) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

11.9.1 目🄵和檔案操作

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])

Copy the contents of the *file-like object* *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

`copyfileobj()` will *not* guarantee that the destination stream has been flushed on completion of the copy. If you want to read from the destination at the completion of the copy operation (for example, reading the contents of a temporary file that has been copied from a HTTP stream), you must ensure that you have called `flush()` or `close()` on the file-like object before attempting to read the destination file.

shutil.**copyfile**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst* in the most efficient way possible. *src* and *dst* are *path-like objects* or path names given as strings.

dst must be the complete target file name; look at `copy()` for a copy that accepts a target directory path. If *src* and *dst* specify the same file, `SameFileError` is raised.

The destination location must be writable; otherwise, an `OSError` exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copyfile`。

在 3.3 版的變更: `IOError` used to be raised instead of `OSError`. Added *follow_symlinks* argument. Now returns *dst*.

在 3.4 版的變更: Raise `SameFileError` instead of `Error`. Since the former is a subclass of the latter, this change is backward compatible.

在 3.8 版的變更: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

exception shutil.**SpecialFileError**

This exception is raised when `copyfile()` or `copytree()` attempt to copy a named pipe.

在 2.7 版被加入.

exception shutil.**SameFileError**

This exception is raised if source and destination in `copyfile()` are the same file.

在 3.4 版被加入.

shutil.**copymode**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, `copymode()` will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see `copystat()` for more information. If `copymode()` cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copymode`。

在 3.3 版的變更: 新增 *follow_symlinks* 引數。

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, *copystat()* also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings.

If *follow_symlinks* is false, and *src* and *dst* both refer to symbolic links, *copystat()* will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the *src* symbolic link, and writing the information to the *dst* symbolic link.

> **備🄵**
>
> Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.
>
> - If `os.chmod in os.supports_follow_symlinks` is `True`, *copystat()* can modify the permission bits of a symbolic link.
> - If `os.utime in os.supports_follow_symlinks` is `True`, *copystat()* can modify the last access and modification times of a symbolic link.
> - If `os.chflags in os.supports_follow_symlinks` is `True`, *copystat()* can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)
>
> On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, *copystat()* will copy everything it can. *copystat()* never returns failure.
>
> 更多資訊請見*os.supports_follow_symlinks*。

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copystat`。

在 3.3 版的變更: Added *follow_symlinks* argument and support for Linux extended attributes.

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be *path-like objects* or strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. If *dst* specifies a file that already exists, it will be replaced. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

copy() copies the file data and the file's permission mode (see *os.chmod()*). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use *copy2()* instead.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copyfile`。

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copymode`。

在 3.3 版的變更: Added *follow_symlinks* argument. Now returns path to the newly created file.

在 3.8 版的變更: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

Identical to *copy()* except that *copy2()* also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never raises an exception because it cannot preserve file metadata.

copy2() uses *copystat()* to copy the file metadata. Please see *copystat()* for more information about platform support for modifying symbolic link metadata.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copyfile`。

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copystat`。

在 3.3 版的變更: Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

在 3.8 版的變更: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

`shutil.`**`ignore_patterns`**(**patterns*)

This factory function creates a function that can be used as a callable for `copytree()`'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

`shutil.`**`copytree`**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

Permissions and times of directories are copied with `copystat()`, individual files are copied using `copy2()`.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed to by the symlink doesn't exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support `os.symlink()`.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `copy2()` is used, but any function that supports the same signature (like `copy()`) can be used.

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.copytree`。

在 3.2 版的變更: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

在 3.3 版的變更: Copy metadata when *symlinks* is false. Now returns *dst*.

在 3.8 版的變更: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

在 3.8 版的變更: 新增 *dirs_exist_ok* 參數。

`shutil.`**`rmtree`**(*path*, *ignore_errors=False*, *onerror=None*, ***, *onexc=None*, *dir_fd=None*)

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onexc* or *onerror* or, if both are omitted, exceptions are propagated to the caller.

This function can support *paths relative to directory descriptors*.

> **備🄵**
>
> On platforms that support the necessary fd-based functions a symlink attack resistant version of *rmtree()* is used by default. On other platforms, the *rmtree()* implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the *rmtree.avoids_symlink_attacks* function attribute to determine which case applies.

If *onexc* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, is the exception that was raised. Exceptions raised by *onexc* will not be caught.

The deprecated *onerror* is similar to *onexc*, except that the third parameter it receives is the tuple returned from *sys.exc_info()*.

> **也參考**
>
> *rmtree* 範例 for an example of handling the removal of a directory tree that contains read-only files.

引發一個附帶引數 `path`、`dir_fd` 的稽核事件 `shutil.rmtree`。

在 3.3 版的變更: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

在 3.8 版的變更: On Windows, will no longer delete the contents of a directory junction before removing the junction.

在 3.11 版的變更: 新增 *dir_fd* 參數。

在 3.12 版的變更: 新增 *onexc* 參數🄵🄵用 *onerror*。

在 3.13 版的變更: `rmtree()` now ignores *FileNotFoundError* exceptions for all but the top-level path. Exceptions other than *OSError* and subclasses of `OSError` are now always propagated to the caller.

`rmtree.`**`avoids_symlink_attacks`**

Indicates whether the current platform and implementation provides a symlink attack resistant version of *rmtree()*. Currently this is only true for platforms supporting fd-based directory access functions.

在 3.3 版被加入.

`shutil.`**`move`**(*src*, *dst*, *copy_function=copy2*)

Recursively move a file or directory (*src*) to another location and return the destination.

If *dst* is an existing directory or a symlink to a directory, then *src* is moved inside that directory. The destination path in that directory must not already exist.

If *dst* already exists but is not a directory, it may be overwritten depending on *os.rename()* semantics.

If the destination is on the current filesystem, then *os.rename()* is used. Otherwise, *src* is copied to the destination using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created as the destination and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments, *src* and the destination, and will be used to copy *src* to the destination if *os.rename()* cannot be used. If the source is a directory, *copytree()* is called, passing it the *copy_function*. The default *copy_function* is *copy2()*. Using *copy()* as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

引發一個附帶引數 `src`、`dst` 的稽核事件 `shutil.move`。

在 3.3 版的變更: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

在 3.5 版的變更: 新增 *copy_function* 關鍵字引數。

在 3.8 版的變更: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See *Platform-dependent efficient copy operations* section.

在 3.9 版的變更: Accepts a *path-like object* for both *src* and *dst*.

`shutil.`**`disk_usage`**(*path*)

Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. *path* may be a file or a directory.

> **備🄵**
>
> On Unix filesystems, *path* must point to a path within a **mounted** filesystem partition. On those platforms, CPython doesn't attempt to retrieve disk usage information from non-mounted filesystems.

在 3.3 版被加入.

在 3.8 版的變更: On Windows, *path* can now be a file or directory.

可用性: Unix, Windows.

`shutil.`**`chown`**(*path*, *user=None*, *group=None*, *, *dir_fd=None*, *follow_symlinks=True*)

Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid; the same applies to *group*. At least one argument is required.

See also `os.chown()`, the underlying function.

引發一個附帶引數 `path`、`user`、`group` 的稽核事件 `shutil.chown`。

可用性: Unix.

在 3.3 版被加入.

在 3.13 版的變更: 新增 *dir_fd* 和 *follow_symlinks* 參數。

`shutil.`**`which`**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to `os.access()`, by default determining if the file exists and is executable.

path is a "`PATH` string" specifying the directories to look in, delimited by `os.pathsep`. When no *path* is specified, the `PATH` environment variable is read from `os.environ`, falling back to `os.defpath` if it is not set.

If *cmd* contains a directory component, `which()` only checks the specified path directly and does not search the directories listed in *path* or in the system's `PATH` environment variable.

On Windows, the current directory is prepended to the *path* if *mode* does not include `os.X_OK`. When the *mode* does include `os.X_OK`, the Windows API `NeedCurrentDirectoryForExePathW` will be consulted to determine if the current directory should be prepended to *path*. To avoid consulting the current working directory for executables: set the environment variable `NoDefaultCurrentDirectoryInExePath`.

Also on Windows, the `PATHEXT` environment variable is used to resolve commands that may not already include an extension. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

This is also applied when *cmd* is a path that contains a directory component:

```
>>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

在 3.3 版被加入.

在 3.8 版的變更: The *bytes* type is now accepted. If *cmd* type is *bytes*, the result type is also *bytes*.

在 3.12 版的變更: On Windows, the current directory is no longer prepended to the search path if *mode* includes `os.X_OK` and WinAPI `NeedCurrentDirectoryForExePathW(cmd)` is false, else the current directory is prepended even if it is already in the search path; `PATHEXT` is used now even when *cmd* includes a directory component or ends with an extension that is in `PATHEXT`; and filenames that have no extension can now be found.

exception shutil.**Error**

This exception collects exceptions that are raised during a multi-file operation. For *copytree()*, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

Platform-dependent efficient copy operations

Starting from Python 3.8, all functions involving a file copy (*copyfile()*, *copy()*, *copy2()*, *copytree()*, and *move()*) may use platform-specific "fast-copy" syscalls in order to copy the file more efficiently (see bpo-33671). "fast-copy" means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in "`outfd.write(infd.read())`".

On macOS fcopyfile is used to copy the file content (not metadata).

Linux 上會使用*os.copy_file_range()* 或*os.sendfile()*。

Solaris 上會使用*os.sendfile()*。

On Windows *shutil.copyfile()* uses a bigger default buffer size (1 MiB instead of 64 KiB) and a *memoryview()*-based variant of *shutil.copyfileobj()* is used.

If the fast-copy operation fails and no data was written in the destination file then shutil will silently fallback on using less efficient *copyfileobj()* function internally.

在 3.8 版的變更.

在 3.14 版的變更: Solaris now uses *os.sendfile()*.

在 3.14 版的變更: Copy-on-write or server-side copy may be used internally via *os.copy_file_range()* on supported Linux filesystems.

copytree 范例

An example that uses the *ignore_patterns()* helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # 🄵有東西被忽略

copytree(source, destination, ignore=_logpath)
```

rmtree 范例

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onexc callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

11.9.2 Archiving operations

在 3.2 版被加入.

在 3.5 版的變更: 新增 *xztar* 格式的支援。

High-level utilities to create and read compressed and archived files are also provided. They rely on the *zipfile* and *tarfile* modules.

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension.

format is the archive format: one of ”zip” (if the *zlib* module is available), ”tar”, ”gztar” (if the *zlib* module is available), ”bztar” (if the *bz2* module is available), ”xztar” (if the *lzma* module is available), or ”zstdtar” (if the *compression.zstd* module is available).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger 必須是與 **PEP 282** 相容的物件，通常是*logging.Logger* 的實例。

verbose 引數未被使用且已🄴用。

引發一個附帶引數 `base_name`、`format`、`root_dir`、`base_dir` 的稽核事件 `shutil.make_archive`。

> **備🄵**
>
> This function is not thread-safe when custom archivers registered with *register_archive_format()* do not support the *root_dir* argument. In this case it temporarily changes the current working directory of the process to *root_dir* to perform archiving.

在 3.8 版的變更: The modern pax (POSIX.1-2001) format is now used instead of the legacy GNU format for archives created with `format="tar"`.

在 3.10.6 版的變更: This function is now made thread-safe during creation of standard `.zip` and tar archives.

shutil.**get_archive_formats**()

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

預設情🄵下*shutil* 提供這些格式：

- *zip*: ZIP file (if the *zlib* module is available).
- *tar*: Uncompressed tar file. Uses POSIX.1-2001 pax format for new archives.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).
- *zstdtar*: Zstandard compressed tar-file (if the *compression.zstd* module is available).

You can register new formats or provide your own archiver for any existing formats, by using *register_archive_format()*.

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to *os.curdir*) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in *make_archive()*).

If *function* has the custom attribute `function.supports_root_dir` set to `True`, the *root_dir* argument is passed as a keyword argument. Otherwise the current working directory of the process is temporarily changed to *root_dir* before calling *function*. In this case *make_archive()* is not thread-safe.

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by *get_archive_formats()* which returns the list of archivers. Defaults to an empty string.

在 3.12 版的變更: Added support for functions supporting the *root_dir* argument.

shutil.**unregister_archive_format**(*name*)

Remove the archive format *name* from the list of supported formats.

shutil.**unpack_archive**(*filename*[, *extract_dir*[, *format*[, *filter*]]])

Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format: one of "zip", "tar", "gztar", "bztar", "xztar", or "zstdtar". Or any other format registered with *register_unpack_format()*. If not provided, *unpack_archive()* will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a *ValueError* is raised.

The keyword-only *filter* argument is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to use `'data'` (default since Python 3.14), unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.)

引發一個附帶引數 `filename`、`extract_dir`、`format` 的稽核事件 `shutil.unpack_archive`。

> **⚠ 警告**
>
> Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

Since Python 3.14, the defaults for both built-in formats (zip and tar files) will prevent the most dangerous of such security issues, but will not prevent *all* unintended behavior. Read the *Hints for further verification* section for tar-specific details.

在 3.7 版的變更: Accepts a *path-like object* for *filename* and *extract_dir*.

在 3.12 版的變更: 新增 *filter* 引數。

`shutil.`**`register_unpack_format`**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive:

- the path of the archive, as a positional argument;
- the directory the archive must be extracted to, as a positional argument;
- possibly a *filter* keyword argument, if it was given to *unpack_archive()*;
- additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.

description can be provided to describe the format, and will be returned by the *get_unpack_formats()* function.

`shutil.`**`unregister_unpack_format`**(*name*)

Unregister an unpack format. *name* is the name of the format.

`shutil.`**`get_unpack_formats`**()

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

預設情🄵下*shutil* 提供這些格式:

- *zip*: ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).
- *zstdtar*: Zstandard compressed tar-file (if the *compression.zstd* module is available).

You can register new formats or provide your own unpacker for any existing formats, by using *register_unpack_format()*.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
```

(繼續下一頁)

(繼續上一頁)

```
-rwx------ tarek/staff      668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff      609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff     1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff      397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff    37192 2010-02-06 18:23:10 ./known_hosts
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use `make_archive()`, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/myarchive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.9.3 Querying the size of the output terminal

`shutil.`**`get_terminal_size`**(*fallback=(columns, lines)*)

Get the size of the terminal window.

For each of the two dimensions, the environment variable, `COLUMNS` and `LINES` respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When `COLUMNS` or `LINES` is not defined, which is the common case, the terminal connected to `sys.__stdout__` is queried by invoking `os.get_terminal_size()`.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)` which is the default size used by many terminal emulators.

The value returned is a named tuple of type `os.terminal_size`.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

在 3.3 版被加入.

在 3.11 版的變更: The `fallback` values are also used if `os.get_terminal_size()` returns zeroes.

也參考

Module *os*
作業系統介面，包括處理比 Python 檔案物件更低階檔案的函式。

Module *io*
Python 的🄴建 I/O 函式庫，包含抽象類🄴和一些具體類🄴 (concrete class)，如檔案 I/O。

🄴建函式*open()*
使用 Python 打開檔案以進行讀寫檔案的標準方法。

CHAPTER 12

資料持久性 (Data Persistence)

本章節所描述的模組支援將 Python 資料以持久形式儲存在磁碟上。*pickle* 和*marshal* 模組可以將許多 Python 資料型🄵轉🄵🄵位元串流，然後再從位元串流重建物件。各種與 DBM 有關的模組支援一系列基於雜🄵的檔案格式，用來儲存字串對字串的對映。

本章節所描述的模組列表🄵:

12.1 `pickle` --- Python 物件序列化

原始碼： Lib/pickle.py

pickle 模組實作的是一個在二進位層級上對 Python 物件進行序列化（serialize）或去序列化（de-serialize）。*"Pickling"* 用於專門指摘將一個 Python 物件轉🄵🄵一個二進位串流的過程，*"unpickling"* 則相反，指的是將一個（來自*binary file* 或*bytes-like object* 的）二進位串流轉🄴回 Python 物件的過程。Pickling（和 unpickling）的過程也可能被稱作"serialization", "marshalling,"[1] 或"flattening"。不過，🄵了避免混淆，本文件將統一稱作封裝（pickling）、拆封（unpickling）。

> **⚠ 警告**
>
> `pickle` 模組🄵**不安全**，切記只拆封你信任的資料。
>
> pickle 封包是有可能被建立來在拆封的時候**執行任意惡意程式碼**的。🄵對不要拆封任何你無法信任其來源、或可能被修改過的 pickle 封包。
>
> 建議你可以使用*hmac* 模組來簽署這個封包，以確保其未被修改過。
>
> 如果你在處理不受信任的資料，其他比較安全的序列化格式（例如*json*）可能會更適合。請參照 See *和 json 的比較* 的🄵明。

12.1.1 和其他 Python 模組的關🄵

和 `marshal` 的比較

Python 有另一個比較原始的序列化模組叫*marshal*，不過其設計目的是🄵了支援 Python 的預編譯功能 `.pyc` 的運作。總地來🄵，請盡可能地使用*pickle*，🄵事不要用*marshal*。

[1] 不要將此模組與*marshal* 模組混淆

pickle 和*marshal* 有幾個明顯不同的地方：

- *pickle* 會記住哪些物件已經被序列化過了，稍後再次參照到這個物件的時候才不會進行重🄵的序列化。*marshal* 🄵有這個功能。

  這對遞🄵物件和物件共用都有影響。遞🄵物件是指包含自我參照的物件。這些情🄵在 marshal 模組中不會被處理，若嘗試使用 marshal 處理遞🄵物件會導致 Python 直譯器崩潰。物件共用發生在序列化的物件階層中、不同位置對同一物件有多個參照時。*pickle* 只會儲存這個被參照的物件一次，🄵確保所有其他參照指向這個主要的版本。共用的物件會保持共用，這對於可變（mutable）物件來🄵非常重要。

- *marshal* 無法序列化使用者自訂的類🄵和的實例。*pickle* 則可以讓使用者儲存🄵還原自訂的類🄵實例，前提是儲存時該類🄵的定義存在於與要被儲存的物件所在的模組中、且可以被引入（import）。

- *marshal* 序列化格式無法保證能在不同版本的 Python 之間移植。因🄵其主要的作用是支援 `.pyc` 檔案的運作，Python 的實作人員會在需要時實作無法前向相容的序列化方式。但只要選擇了相容的 pickle 協定，且處理了 Python 2 和 Python 3 之間的資料類型差🄵，*pickle* 序列化協定能保證在不同 Python 版本間的相容性。

和 `json` 的比較

pickle 協定和 JSON (JavaScript Object Notation) 有一些根本上的不同：

- JSON 以文字形式作🄵序列化的輸出（輸出 unicode 文字，但大多數又會被編碼🄵 `UTF-8`），而 pickle 則是以二進位形式作🄵序列化的輸出；
- JSON 是人類可讀的，而 pickle 則無法；
- JSON 具有高互通性（interoperability）且在 Python 以外的環境也被大量利用，但 pickle 只能在 Python 🄵使用。
- 預設狀態下的 JSON 只能紀🄵一小部份的 Python 🄵建型🄵，且無法紀🄵自訂類🄵；但透過 Python 的自省措施，pickle 可以紀🄵🄵大多數的 Python 型🄵（其他比較🄵雜的狀🄵也可以透過實作*specific object APIs* 來解🄵）；
- 去序列化不安全的 JSON 不會🄵生任意程式執行的風險，但去序列化不安全的 pickle 會。

也參考

json module: 是標准函式庫的一部分，可讓使用者進行 JSON 的序列化與去序列化。

12.1.2 資料串流格式

pickle 使用的資料格式是針對 Python 而設計的。好處是他不會受到外部標准（像是 JSON，無法紀🄵指標共用）的限制；不過這也代表其他不是 Python 的程式可能無法重建 pickle 封裝的 Python 物件。

以預設設定來🄵，*pickle* 使用相對緊🄵的二進位形式來儲存資料。如果你需要盡可能地縮小檔案大小，你可以壓縮封裝的資料。

pickletools 含有工具可分析*pickle* 所🄵生的資料流。*pickletools* 的源始碼詳細地記載了所有 pickle 協定的操作碼（opcode）。

截至目前🄵止，共有六種不同版本的協定可用於封裝 pickle。數字越大版本代表你需要使用越新的 Python 版本來拆封相應的 pickle 封裝。

- 版本 0 的協定是最初「人類可讀」的版本，且可以向前支援早期版本的 Python。
- 版本 1 的協定使用舊的二進位格式，一樣能向前支援早期版本的 Python。
- 版本 2 的協定在 Python 2.3 中初次被引入。其可提供更高效率的*new-style classes* 封裝過程。請參🄵 **PEP 307** 以了解版本 2 帶來的改進。
- 版本 3 的協定在 Python 3.0 被新增。現在能支援封裝*bytes* 的物件且無法被 2.x 版本的 Python 拆封。在 3.0 ~ 3.7 的 Python 預設使用 3 版協定。

- 版本 4 的協定在 Python 3.4 被新增。現在能支援超大物件的封裝、更多種型🄴的物件以及針對部份資料格式的儲存進行最佳化。從 Python 3.8 到 3.13，預設使用第 4 版協定。請參🄵 **PEP 3154** 以了解第 4 版協定改進的細節。
- 版本 5 的協定在 Python 3.8 被新增。現在能支援帶外資料（Out-of-band data）🄴加速帶🄴資料的處理速度。自 3.14 起這是預設使用的協定。請參🄵 **PEP 574** 以了解第 5 版協定改進的細節。

備🄵

資料序列化是一個比資料持久化更早出現的概念；雖然*pickle* 可以讀寫檔案物件，但它🄴不處理命名持久物件的問題，也不處理對持久物件的🄴行存取、一個更棘手的問題。*pickle* 模組可以將🄴雜物件轉🄴成位元組串流，也可以將位元組串流轉🄴回具有相同原始🄴部結構的物件。對這些位元組串流最常見的處理方式是將它們寫入檔案中，但也可以將它們透過網路傳送或儲存在一個資料庫中。*shelve* 模組提供了一個簡單的介面來讓使用者在 DBM 風格的資料庫檔案中對物件進行封裝和拆封的操作。

12.1.3 模組介面

想要序列化一個物件，你只需要呼叫*dumps()* 函式。而當你想要去序列化一個資料流時，你只需要呼叫*loads()* 即可。不過，若你希望能各自對序列化和去序列化的過程中有更多的掌控度，你可以自訂一個*Pickler* 或*Unpickler* 物件。

pickle 模組提供以下常數：

pickle.**HIGHEST_PROTOCOL**

一個整數，表示可使用的最高協定版本。這個值可作🄵 *protocol* 的數值傳給*dump()* 和*dumps()* 函式以及*Pickler* 建構式。

pickle.**DEFAULT_PROTOCOL**

一個整數，用於 pickle 的預設協定版本。可能小於*HIGHEST_PROTOCOL*。目前預設協定是 5，在 Python 3.8 中引入，與之前的版本不相容。此版本引入了對 out-of-band buffer 的支援，其中 **PEP 3118** 相容的資料可以與主要 pickle 串流分開傳輸。

在 3.0 版的變更: 預設協定版本🄵 3。

在 3.8 版的變更: 預設協定版本🄵 4。

在 3.14 版的變更: 預設協定是 5。

pickle 模組提供下列函式來簡化封裝的過程：

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

將被封裝成 pickle 形式的物件 *obj* 寫入到已開🄵的*file object* *file*。這等效於 `Pickler(file, protocol).dump(obj)`。

引數 *file*、*protocol*、*fix_imports* 和 *buffer_callback* 的意義與*Pickler* 建構式中的相同。

在 3.8 版的變更: 新增 *buffer_callback* 引數。

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

將被封裝🄵 pickle 形式的物件 *obj* 以*bytes* 類🄴回傳，而非寫入進檔案。

引數 *protocol*、*fix_imports* 和 *buffer_callback* 的意義和*Pickler* 建構式中的相同。

在 3.8 版的變更: 新增 *buffer_callback* 引數。

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

從已開🄴的檔案物件 *file* 中讀取已序列化的物件，🄵傳回其重建後的物件階層。這相當於呼叫 `Unpickler(file).load()`。

模組會自動偵測 pickle 封包所使用的協定版本，所以無須另外指定。超出 pickle 封包表示範圍的位元組將被忽略。

引數 *file*、*fix_imports*、*encoding*、*errors*、*strict* 和 *buffers* 的意義和*Unpickler* 建構式中的相同。

在 3.8 版的變更: 新增 *buffer* 引數。

pickle.**loads**(*data*, */*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

回傳從 *data* 的 pickle 封包重建後的物件階層。*data* 必須是一個*bytes-like object*。

模組會自動偵測 pickle 封包所使用的協定版本，所以無須另外指定。超出 pickle 封包表示範圍的位元組將被忽略。

引數 *fix_imports*、*encoding*、*errors*、*strict* 和 *buffers* 的意義與*Unpickler* 建構式所用的相同。

在 3.8 版的變更: 新增 *buffer* 引數。

pickle 模組定義了以下三種例外:

exception pickle.**PickleError**

繼承*Exception* 類🄵。一個在封裝或拆封時遭遇其他例外時通用的基底類🄵。

exception pickle.**PicklingError**

當*Pickler* 遭遇無法封裝物件時會引發的例外。繼承*PickleError* 類🄵。

請參🄵哪些物件能或不能被封裝、拆封？ 以了解哪些物件是可以被封裝的。

exception pickle.**UnpicklingError**

拆封物件時遇到問題（如資料🄵損或違反安全性原則等）所引發的意外。繼承自*PickleError* 類🄵。

拆封的時候還是可能會遭遇其他不在此列的例外（例如：AttributeError、EOFError、ImportError、或 IndexError），請注意。

引入模組*pickle* 時會帶來三個類🄵：*Pickler*、*Unpickler* 和*PickleBuffer*:

class pickle.**Pickler**(*file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

接受一個用以寫入 pickle 資料流的二進位檔案。

可選引數 *protocol* 接受整數，用來要求封裝器（pickler）使用指定的協定；支援從 0 版起到*HIGHEST_PROTOCOL* 版的協定。如未指定，則預設🄵*DEFAULT_PROTOCOL*。若指定了負數，則視🄵選擇*HIGHEST_PROTOCOL*。

引數 *file* 必須支援可寫入單一位元組引數的 write() 方法。只要滿足此條件，傳入的物件可以是一個硬碟上二進位檔案、一個*io.BytesIO* 實例或任何其他滿足這個介面要求的物件。

若 *fix_imports* 設🄵 true 且 *protocol* 版本小於 3，本模組會嘗試將 Python 3 的新模組名稱轉🄵🄵 Python 2 所支援的舊名，以讓 Python 2 能正確地讀取此資料流。

如果 *buffer_callback* 是 None（預設值），緩衝區的視圖會作🄵 pickle 封裝串流的一部分被序列化進 *file* 中。

如果 *buffer_callback* 不是 None，則它可以被多次呼叫🄵回傳一個緩衝區的視圖。如果回呼函式回傳一個假值（例如 None），則所給的緩衝區將被視🄵帶外資料；否則，該緩衝區將被視🄵 pickle 串流的帶🄵資料被序列化。

如果 *buffer_callback* 不是 None 且 *protocol* 是 None 或小於 5 則會報錯。

在 3.8 版的變更: 新增 *buffer_callback* 引數。

dump(*obj*)

將已封裝（pickled）的 *obj* 寫入已在建構式中開🄵的對應檔案。

persistent_id(*obj*)

預設不進行任何動作。這是一種抽象方法，用於讓後續繼承這個類🄵的物件可以覆寫本方法函式。

如果*persistent_id()* 回傳 None，則 *obj* 會照一般的方式進行封裝（pickling）。若回傳其他值，則*Pickler* 會將該值作🄵 *obj* 的永久識🄵碼回傳。此永久識🄵碼的意義應由*Unpickler.persistent_load()* 定義。請注意*persistent_id()* 回傳的值本身不能擁有自己的永久識🄵碼。

關於細節與用法範例請見外部物件持久化。

在 3.13 版的變更: 在 C 的 `Pickler` 實作中的增加了這個方法的預設實作。

dispatch_table

封裝器（pickler）物件含有的的調度表是一個縮🄵函式（reduction function）的🄵🄵表，可以使用*copyreg.pickle()* 來宣告這類縮🄵函式。它是一個以類🄵🄵鍵、還原函式🄵值的映射表。縮🄵函式應准備接收一個對應類🄵的引數，🄵應遵循與*__reduce__()* 方法相同的介面。

預設情🄵下，封裝器（pickler）物件不會有*dispatch_table* 屬性，而是會使用由*copyreg* 模組管理的全域調度表。不過，若要自訂某個封裝器（pickler）物件的序列化行🄵，可以將*dispatch_table* 屬性設置🄵類字典物件。另外，如果*Pickler* 的子類🄵具有*dispatch_table* 屬性，那🄵這個屬性將作🄵該子類🄵實例的預設調度表。

關於用法範例請見調度表。

在 3.3 版被加入.

reducer_override(*obj*)

一個可以在*Pickler* 子類🄵中被定義的縮🄵器（reducer）。這個方法的優先度高於任何其他分派表 中的縮🄵器。他應該要有和*__reduce__()* 方法相同的函式介面，且可以可選地回傳*NotImplemented* 以後備（fallback）使用分派表 中登🄵的縮🄵方法來封裝 `obj`。

請查🄵針對型🄵、函式或特定物件定🄵縮🄵函式 來參考其他較詳細的範例。

在 3.8 版被加入.

fast

已🄵用。如果設置🄵 true，將🄵用快速模式。快速模式會停用備忘（memo），因此能透過不🄵生多余的 PUT 操作碼（OpCode）來加速封裝過程。它不應被用於自我參照物件，否則將導致*Pickler* 陷入無限遞🄵。

使用*pickletools.optimize()* 以獲得更緊🄵的 pickle 輸出。

clear_memo()

Clears the pickler's "memo".

The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects are pickled by reference and not by value. This method is useful when re-using picklers.

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

這個物件接受一個二進位檔案 *file* 來從中讀取 pickle 資料流。

協定版本號會被自動偵測，所以不需要在這邊手動輸入。

參數 *file* 必須擁有三個方法，分🄵是接受整數作🄵引數的 read() 方法、接受緩衝區作🄵引數的 readinto() 方法以及不需要引數的 readline() 方法，如同在*io.BufferedIOBase* 的介面一樣。因此，*file* 可以是一個以二進位讀取模式開🄵的檔案、一個*io.BytesIO* 物件、或任何符合此介面的自訂物件。

可選引數 *fix_imports*、*encoding* 和 *errors* 用來控制 Python 2 pickle 資料的相容性支援。如果 *fix_imports* 🄵 true，則 pickle 模組會嘗試將舊的 Python 2 模組名稱映射到 Python 3 中使用的新名稱。*encoding* 和 *errors* 告訴 pickle 模組如何解碼由 Python 2 pickle 封裝的 8 位元字串實例；*encoding* 和 *errors* 預設分🄵🄵'ASCII' 和'strict'。*encoding* 可以設定🄵'bytes' 以將這些 8 位元字串實例讀🄵位元組物件。而由 Python 2 封裝的 NumPy 陣列、*datetime*、*date* 和*time* 的實例則必須使用 `encoding='latin1'` 來拆封。

如果 *buffers* 是 `None`（預設值），那🄵去序列化所需的所有資料都必須已經包含在 pickle 串流中。這意味著當初在建立對應的*Pickler* 時（或在呼叫*dump()* 或*dumps()* 時）*buffer_callback* 引數必須🄵 `None`。

如果 *buffers* 不是 `None`，則其應該是一個可🄵代物件，🄵含數個支援緩衝區的物件，🄵且每當 pickle 串流引用一個帶外緩衝區視圖時將會被照順序消耗。這些緩衝資料當初建立時應已按照順序給定於 Pickler 物件中的 *buffer_callback*。

在 3.8 版的變更: 新增 *buffer* 引數。

load()

開🄵先前被傳入建構子的檔案，從中讀取一個被 pickle 封裝的物件，🄵回傳重建完成的物件階層。超過 pickle 表示範圍的位元組會被忽略。

persistent_load(*pid*)

預設會引發*UnpicklingError* 例外。

若有定義，則*persistent_load()* 將回傳符合持久化識🄵碼 *pid* 的物件。如果遭遇了無效的持久化識🄵碼，則會引發*UnpicklingError*。

關於細節與用法範例請見外部物件持久化。

在 3.13 版的變更: 在 C 的 `Unpickler` 實作中的增加了這個方法的預設實作。

find_class(*module*, *name*)

如有需要將引入 *module*，🄵從中回傳名🄵 *name* 的物件，這🄵的 *module* 和 *name* 引數接受的輸入是*str* 物件。注意，雖然名稱上看起來不像，但*find_class()* 亦可被用於尋找其他函式。

子類🄵可以覆寫此方法以控制可以載入哪些類型的物件、以及如何載入它們，從而🄵在地降低安全性風險。詳情請參考限制全域物件。

引發一個附帶引數 `module`、`name` 的稽核事件 `pickle.find_class`。

class pickle.**PickleBuffer**(*buffer*)

一個表示了含有可封裝資料緩衝區的包裝函式（wrapper function）。*buffer* 必須是一個 提供緩衝區的物件，例如一個類位元組物件 或 N 維陣列。

PickleBuffer 本身就是一個提供緩衝區的物件，所以是能🄵將其提供給其它「預期收到含有緩衝物件的 API」的，比如*memoryview*。

PickleBuffer 物件僅能由 5 版或以上的 pickle 協定進行封裝。該物件亦能被作🄵帶外資料來進行帶外資料序列化

在 3.8 版被加入.

raw()

回傳此緩衝區底層記憶體區域的*memoryview*。被回傳的物件是一個（在 C 語言的 formatter 格式中）以 `B` (unsigned bytes) 二進位格式儲存、一維且列連續（C-contiguous）的 memoryview。如果緩衝區既不是列連續（C-contiguous）也不是行連續（Fortran-contiguous）的，則會引發*BufferError*。

release()

釋放 PickleBuffer 物件現正曝光中的緩衝區。

12.1.4 哪些物件能或不能被封裝、拆封？

下列型🄵可以被封裝:

- 🄵建常數（`None`、`True`、`False`、`Ellipsis` 和*NotImplemented*）;
- 整數、浮點數和🄵數;
- 字串、位元組物件、位元組陣列;
- 元組（tuple）、串列（list）、集合（set）和僅含有可封裝物件的字典;
- 在模組最表面的層級就能被存取的函式（🄵建或自訂的皆可，不過僅限使用 `def` 定義的函式，`lambda` 函式不適用）;
- 在模組最表面的層級就能被存取的類🄵;
- 實例，只要在呼叫了*__getstate__()* 後其回傳值全都是可封裝物件。（詳情請參🄵*Pickling* 類🄵實例）。

嘗試封裝無法封裝的物件會引發*PicklingError* 例外；注意當這種情🄵發生時，可能已經有未知數量的位元組已被寫入到檔案。嘗試封裝深度遞🄵的資料結構可能會導致其超出最大遞🄵深度，在這種情🄵下會引發*RecursionError* 例外。你可以（小心地）使用*sys.setrecursionlimit()* 來提高此上限。

請注意，函式（🄵建及自訂兩者皆是）是依據完整的限定名稱來封裝，而非依其值。[2] 這意味著封裝時只有函式名稱、所屬的模組和所屬的類🄵名稱會被封裝。函式本身的程式碼及其附帶的任何屬性均不會被封裝。因此，在拆封該物件的環境中，定義此函式的模組必須可被引入，且該模組必須包含具此命名之物件，否則將引發例外。[3]

同樣情🄵，類🄵是依照其完整限定名稱來進行封裝，因此在進行拆封的環境中會具有同上的限制。類🄵中的程式碼或資料皆不會被封裝，因此在以下範例中，注意到類🄵屬性 `attr` 在拆封的環境中不會被還原:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

這些限制就是可封裝的函式和類🄵必須被定義在模組頂層的原因。

同樣地，當類🄵實例被封裝時，它所屬類🄵具有的程式碼和資料不會被一起封裝。只有實例資料本身會被封裝。這是有意而🄵的，因🄵如此你才可以在類🄵中修正錯誤或新增其他方法，且於此同時仍能🄵載入使用較早期版本的類🄵所建立的物件實例。如果你預計將有長期存在的物件、且該物件將經歷許多版本的更替，你可以在物件中存放一個版本號，以便未來能透過 *__setstate__()* 方法來進行適當的版本轉🄵。

12.1.5 Pickling 類🄵實例

在這一個章節，我們會講述如何封裝或拆封一個物件實例的相關機制，以方便你進行自訂。

大部分的實例不需要額外的程式碼就已經是可封裝的了。在這樣的預設狀🄵中，pickle 模組透過自省機制來取得類🄵及其實例的屬性。當類🄵實例被拆封時，其 `__init__()` 方法通常不會被呼叫。預設行🄵首先會建立一個未初始化的實例，然後還原紀🄵中的屬性。以下程式碼的實作展示了前述行🄵:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

被封裝的目標類🄵可以提供一個或數個下列特殊方法來改變 pickle 的預設行🄵:

`object.`**`__getnewargs_ex__`**`()`

在第 2 版協定或更新的版本中，有實作 *__getnewargs_ex__()* 方法的類🄵，可以🄵定在拆封時要傳遞給 `__new__()` 方法的值。該方法必須回傳一個 `(args, kwargs)` 的組合，其中 *args* 是一個位置引數的元組（tuple），*kwargs* 是一個用於建構物件的命名引數字典。這些資訊將在拆封時傳遞給 `__new__()` 方法。

如果目標類🄵的方法 `__new__()` 需要僅限關鍵字的參數時，你應該實作此方法。否則，🄵了提高相容性，建議你改🄵實作 *__getnewargs__()*。

在 3.6 版的變更: 在第 2、3 版的協定中現在改🄵使用 *__getnewargs_ex__()*。

`object.`**`__getnewargs__`**`()`

此方法與 *__getnewargs_ex__()* 的目的一樣，但僅支援位置參數。它必須回傳一個由傳入引數所組成的元組（tuple）`args`，這些引數會在拆封時傳遞給 `__new__()` 方法。

當有定義 *__getnewargs_ex__()* 的時候便不會呼叫 *__getnewargs__()*。

在 3.6 版的變更: 在 Python 3.6 之前、版本 2 和版本 3 的協定中，會呼叫 *__getnewargs__()* 而非 *__getnewargs_ex__()*。

[2] 這就是🄵什🄵 `lambda` 函式無法被封裝：所有 `lambda` 函式共享相同的名稱：`<lambda>`。
[3] 引發的例外應該是 *ImportError* 或 *AttributeError*，但也可能是其他例外。

object.**__getstate__**()

目標類別可以透過覆寫方法 *__getstate__()* 進一步影響其實例被封裝的方式。封裝時，呼叫該方法所回傳的物件將作為該實例的內容被封裝、而非一個預設狀態。以下列出幾種預設狀態：

- 有 __dict__ 和 __slots__ 實例的類別，其預設狀態為 None。
- 有 __dict__ 實例、但有 __slots__ 實例的類別，其預設狀態為 self.__dict__。
- 有 __dict__ 和 __slots__ 實例的類別，其預設狀態是一個含有兩個字典的元組（tuple），該二字典分別為 self.__dict__ 本身，和紀錄欄位（slot）名稱和值對應關係的字典（只有含有值的欄位（slot）會被紀錄其中）。
- 有 __dict__ 但有 __slots__ 實例的類別，其預設狀態是一個二元組（tuple），元組中的第一個值是 None，第二個值則是紀錄欄位（slot）名稱和值對應關係的字典（與前一項提到的字典是同一個）。

在 3.11 版的變更: 在 *object* 類中增加預設的 __getstate__() 實作。

object.**__setstate__**(*state*)

在拆封時，如果類定義了 *__setstate__()*，則會使用拆封後的狀態呼叫它。在這種情下，紀狀態的物件不需要是字典（dictionary）。否則，封裝時的狀態紀必須是一個字典，其紀的項目將被賦值給新實例的字典。

> **備**
>
> 如果 *__reduce__()* 在封裝時回傳了 None 狀態，則拆封時就不會去呼叫 *__setstate__()*。

參處紀大量狀態的物件 以了解 *__getstate__()* 和 *__setstate__()* 的使用方法。

> **備**
>
> 在拆封時，某些方法如 __getattr__()、__getattribute__() 或 __setattr__() 可能會在建立實例時被呼叫。如果這些方法依賴了某些實例部的不變性，則應實作 __new__() 以建立此不變性，因在拆封實例時不會呼叫 __init__()。

如稍後所演示，pickle 不直接使用上述方法。這些方法實際上是實作了 *__reduce__()* 特殊方法的拷貝協定（copy protocol）。拷貝協定提供了統一的介面，以檢索進行封裝及物件時所需的資料。[4]

直接在類中實作 *__reduce__()* 雖然功能大但容易導致出錯。因此，設計類者應盡可能使用高階介面（例如，*__getnewargs_ex__()*、*__getstate__()* 和 *__setstate__()*）。不過，我們也將展示一些特例狀，在這些狀中，使用 __reduce__() 可能是唯一的選擇、是更有效率的封裝方法或二者兼備。

object.**__reduce__**()

目前的介面定義如下。*__reduce__()* 方法不接受引數，且應回傳一個字串或一個元組（元組一般而言是較佳的選擇；所回傳的物件通常稱「縮值」）。

如果回傳的是字串，該字串應被解讀一個全域變數的名稱。它應是該物件相對其所在模組的本地名稱；pickle 模組會在模組命名空間中尋找，以確定該物件所在的模組。這種行通常對於單例物件特有用。

當回傳一個元組時，其長度必須介於兩至六項元素之間。可選項可以被省略，或者其值可以被設 None。各項物件的語意依序：

- 一個將會被呼叫來建立初始版本物件的可呼叫物件。
- 一個用於傳遞引數給前述物件的元組。如果前述物件不接受引數輸入，則你仍應在這給定一個空元組。

[4] *copy* 模組使用此協定進行淺層及深層操作。

- 可選項。物件狀態。如前所述，會被傳遞給該物件的*__setstate__()* 方法。如果該物件🄵有實作此方法，則本值必須是一個字典，且其將會被新增到物件的 __dict__ 屬性中。
- 可選項。一個用來提供連續項目的🄵代器（而非序列）。這些項目將個🄵透過 `obj.append(item)` 方法或成批次地透過 `obj.extend(list_of_items)` 方法被附加到物件中。主要用於串列（list）子類🄵，但只要其他類🄵具有相應的*append()* 和*extend()* 方法以及相同的函式簽章（signature）就也可以使用。（是否會呼叫 `append()` 或 `extend()` 方法將取🄵於所選用的 pickle 協定版本以及要附加的項目數量，因此必須同時支援這兩種方法。）
- 可選項。一個🄵生連續鍵值對的🄵代器（不是序列）。這些項目將以 `obj[key] = value` 方式被儲存到物件中。主要用於字典（dictionary）子類🄵，但只要有實現了 `__setitem__()` 的其他類🄵也可以使用。
- 可選項。一個具有 `(obj, state)` 函式簽章（signature）的可呼叫物件。該物件允許使用者以可編寫的邏輯，而不是物件 `obj` 預設的*__setstate__()* 🄵態方法去控制特定物件的狀態更新方式。如果這個物件不是 `None`，這個物件的呼叫優先權將優於物件 `obj` 的*__setstate__()*。

  在 3.8 版被加入: 加入第六個可選項（一個 `(obj, state)` 元組）。

`object.`**`__reduce_ex__`**(*protocol*)

另外，你也可以定義一個*__reduce_ex__()* 方法。唯一的不同的地方是此方法只接受協定版本（整數）作🄵參數。當有定義本方法時，pickle 會優先呼叫它而不是*__reduce__()* 。此外，呼叫*__reduce__()* 時也會自動變成呼叫這個變體版本。此方法主要是🄵了向後相容的舊的 Python 版本而存在。

外部物件持久化

🄵了方便物件持久化，*pickle* 模組支援對被封裝資料串流以外的物件參照。被參照的物件是透過一個持久化 ID 來參照的，這個 ID 應該要是字母數字字元（alphanumeric）組成的字串（協定 0）[5] 或者是任意的物件（任何較新的協定）。

pickle 🄵有定義要如何解🄵或分派這個持久化 ID 的問題；故其處理方式有賴使用者自行定義在封裝器（pickler）以及拆封器（unpickler）中。方法的名稱各自🄵*persistent_id()* 和*persistent_load()*。

要封裝具有外部持久化 ID 的物件，封裝器（pickler）必須擁有一個自訂的方法*persistent_id()*，這個方法將接收一個物件作🄵參數，🄵回傳 `None` 或該物件的持久化 ID。當回傳 `None` 時，封裝器會正常地封裝該物件。當回傳一個持久化 ID 字串時，封裝器會封裝該物件🄵加上一個標記，讓拆封器（unpickler）能識🄵它是一個持久化 ID。

要拆封外部物件，拆封器（unpickler）必須有一個自訂的*persistent_load()* 方法，該方法應接受一個持久化 ID 物件，🄵回傳相對應的物件。

以下是一個完整的範例，用以🄵明如何使用持久化 ID 來封裝具外部參照的物件。

```
# 展示如何使用持久化 ID 來封裝外部物件的簡單範例

import pickle
import sqlite3
from collections import namedtuple

# 代表資料庫中紀🄵的一個簡易類🄵
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # 我們派發出一個持久 ID，而不是像一般類🄵實例那樣封裝 MemoRecord。
        if isinstance(obj, MemoRecord):
            # 我們的持久 ID 就是一個元組，🄵面包含一個標🄵和一個鍵，指向資料庫中的特定紀🄵。
            return ("MemoRecord", obj.key)
        else:
```

(繼續下一頁)

[5] 協定 0 中限制僅能使用英文字母或數字字元來分配持久化 ID 是因🄵持久化 ID 是由🄵行符號所分隔的。因此，如果持久化 ID 中出現任何形式的🄵行字元，將導致封裝資料變得無法讀取。

(繼續上一頁)

```
            # 如果 obj 🄵有持久 ID，則回傳 None。這表示 obj 像平常那樣封裝即可。
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # 每當遇到持久 ID 時，此方法都會被呼叫。
        # pid 是 DBPickler 所回傳的元組。
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # 從資料庫中抓取所引用的紀🄵🄵回傳。
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # 如果無法回傳正確的物件，則必須引發錯誤。
            # 否則 unpickler 會誤認🄵 None 是持久 ID 所引用的物件。
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # 初始化資料庫。
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # 抓取要封裝的紀🄵。
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # 使用我們自訂的 DBPickler 來保存紀🄵。
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("被封裝的紀🄵:")
    pprint.pprint(memos)

    # 更新一筆紀🄵（測試用）。
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # 從 pickle 資料流中載入紀🄵。
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("已拆封的紀🄵:")
    pprint.pprint(memos)
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == '__main__':
    main()
```

調度表

如果你希望在不干擾其他物件正常封裝的前提下建立一個針對特定物件的封裝器，你可建立一個有私密調度表的封裝器。

由*copyreg* 模組管理的全域調度表可以 `copyreg.dispatch_table` 呼叫。你可以透過這個方式來基於原始 `copyreg.dispatch_table` 建立一個修改過的版本，作🄵你的專屬用途的調度表。

舉例來🄵：

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

建立了一個*pickle.Pickler*，其中含有專門處🄴 `SomeClass` 類🄴的專屬調度表。此外，你也可以寫作：：

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

這樣可🄵生相似的結果，唯一不同的是往後所有 `MyPickler` 預設都會使用這個專屬調度表。最後，如果將程式寫🄵：：

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

則會改變*copyreg* 模組🄴建、所有使用者共通的調度表。

處🄵紀🄵大量狀態的物件

以下的範例展示了如何修改針對特定類🄴封裝時的行🄵。下面的 `TextReader` 類🄴會開🄴一個文字檔案，🄴在每次呼叫其 `readline()` 方法時返回目前行編號與該行🄴容。如果 `TextReader` 實例被封裝，所有除了檔案物件之外的屬性成員都會被保存。在該實例被拆封時，檔案將被重新開🄴，🄴從上次的位置繼續讀取。這個行🄵的達成是透過 `__setstate__()` 和 `__getstate__()` 方法來實作的。：

```
class TextReader:
    """列出文字檔案中的行🄵對其進行編號。"""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
```

(繼續下一頁)

(繼續上一頁)

```
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # 從 self.__dict__ 中🄵🄵物件的狀態。包含了所有的實例屬性。
        # 使用 dict.copy() 方法以避免修改原始狀態。
        state = self.__dict__.copy()
        # 移除不可封裝的項目。
        del state['file']
        return state

    def __setstate__(self, state):
        # 恢復實例屬性（即 filename 和 lineno）。
        self.__dict__.update(state)
        # 恢復到先前開🄵了檔案的狀態。🄵此，我們需要重新開🄵它🄵一直讀取到行數編號相同。
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # 存檔。
        self.file = file
```

可以這樣實際使用：：

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 針對型🄵、函式或特定物件定🄵縮🄵函式

在 3.8 版被加入.

有時候，*dispatch_table* 的彈性空間可能不🄵。尤其當我們想要使用型🄵以外的方式來判斷如何使用自訂封裝、或者我們想要自訂特定函式和類🄵的封裝方法時。

如果是這樣的話，可以繼承*Pickler* 類🄵🄵實作一個*reducer_override()* 方法。此方法可以回傳任意的縮🄵元組（參🄵*__reduce__()*）、也可以回傳*NotImplemented* 以使用後備的原始行🄵方案。

如果*dispatch_table* 和*reducer_override()* 都被定義了的話，*reducer_override()* 的優先度較高。

備🄵

出於效能考量，處🄵以下物件可能不會呼叫*reducer_override()*：None、True、False，以及*int*、*float*、*bytes*、*str*、*dict*、*set*、*frozenset*、*list* 和*tuple* 的實例。

以下是一個簡單的例子，我們示範如何允許封裝和重建給定的類🄵：：

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """MyClass 的自訂縮🄵函式。"""
        if getattr(obj, "__name__", None) == "MyClass":
```

(繼續下一頁)

(繼續上一頁)

```
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # 遭遇其他物件，則使用一般的縮🄴方式
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 帶外（Out-of-band）資料緩沖區

在 3.8 版被加入.

pickle 模組會被用於用於傳輸龐大的資料。此時，將🄵🄵記憶體的次數降到最低以保持效能變得很重要。然而，*pickle* 模組的正常操作過程中，當它將物件的圖狀結構（graph-like structure）轉🄵🄵連續的位元組串流時，本質上就涉及將資料🄵🄵到封裝流以及從封裝流🄵🄵資料。

如果供給者（被傳遞物件的型🄴的實作）與消費者（資訊交🄴系統的實作）都支援由 pickle 協定 5 或更高版本提供的帶外傳輸功能，則可以避免此一先天限制。

供給者 API

要封裝的大型資料物件，則必須實作一個針對 5 版協定及以上的*__reduce_ex__()* 方法，該方法應回傳一個*PickleBuffer* 實例來處理任何大型資料（而非回傳如*bytes* 物件）。

一個*PickleBuffer* 物件指示了當下底層的緩衝區狀態適合進行帶外資料傳輸。這些物件仍然相容*pickle* 模組的一般使用方式。消費者程式也可以選擇介入，指示*pickle* 他們將自行處理這些緩衝區。

消費者 API

一個資訊交🄴系統可以🄴定要自行處🄴序列化物件圖時🄴生的*PickleBuffer* 物件。

傳送端需要傳遞一個呼叫緩衝區的回呼函式給*Pickler*（或*dump()* 或*dumps()* 函式）的 *buffer_callback* 引數，使每次生成*PickleBuffer* 時，該物件在處理物件圖時能被呼叫。除了一個簡易標記以外，由 *buffer_callback* 累積的緩衝區資料不會被🄵🄵到 pickle 串流中。

接收端需要傳遞一個緩衝區物件給*Unpickler*（或*load()* 或*loads()* 函式）的 *buffers* 引數。該物件須是一個可🄴代的（iterable）緩衝區（buffer）物件，其中包含傳遞給 *buffer_callback* 的緩衝區物件。這個可🄴代物件的緩衝區順序應該與它們當初被封裝時傳遞給 *buffer_callback* 的順序相同。這些緩衝區將提供物件重建所需的資料，以使重建器能還原出那個當時🄴生了*PickleBuffer* 的物件。

在傳送與接收端之間，通訊系統可以自由實作轉移帶外緩衝區資料的機制。該機制可能可以利用共用記憶體機制或根據資料類型特定的壓縮方式來最佳化執行速度。

范例

這一個簡單的範例展示了如何實作一個可以參與帶外緩衝區封裝的*bytearray* 子類🄴：：

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
```

(繼續下一頁)

(繼續上一頁)

```
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer 在 pickle 協定 <= 4 時禁止使用。
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # 取得對原始緩衝區物件的控制
            obj = m.obj
            if type(obj) is cls:
                # 若原本的緩衝區物件是 ZeroCopyByteArray，則直接回傳。
                return obj
            else:
                return cls(obj)
```

如果型🄵正確，重建器（_reconstruct 類🄵方法）會回傳當時提供緩衝區的物件。這個簡易實作可以模擬一個無🄵🄵行🄵的重建器。

在使用端，我們可以用一般的方式封裝這些物件，當我們拆封時會得到一個原始物件的副本：：

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: 曾進行過🄵🄵運算
```

但如果我們傳一個 *buffer_callback* 🄵在去序列化時正確回傳積累的緩衝資料，我們就能拿回原始的物件：：

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: 🄵有進行過🄵🄵
```

此範例是因🄵受限於*bytearray* 會自行分配記憶體：你無法建立以其他物件的記憶體🄵基礎的*bytearray* 實例。不過第三方資料型態（如 NumPy 陣列）則可能🄵有這個限制，而允許在不同程序或系統之間傳輸資料時使用零拷貝封裝（或🄵可能地🄵少拷貝次數）。

也參考

PEP 574 -- 第 5 版 Pickle 協定的帶外資料（out-of-band data）處🄵

12.1.8 限制全域物件

預設情🄵下，拆封過程將會引入任何在 pickle 資料中找到的類🄵或函式。對於許多應用程式來🄵，這種行🄵是不可接受的，因🄵它讓拆封器能🄵引入🄵執行任意程式碼。請參見以下 pickle 資料流在載入時的行🄵：：

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

在這個例子中，拆封器會引入*os.system()* 函式，然後執行命令「echo hello world」。雖然這個例子是無害的，但不難想像可以這個方式輕易執行任意可能對系統造成損害的命令。

基於以上原因，你可能會希望透過自訂*Unpickler.find_class()* 來控制哪些是能🄵被拆封的🄵容。與其名稱字面意義暗示的不同，實際上每當你請求一個全域物件（例如，類🄵或函式）時，就會呼叫*Unpickler.find_class()*。因此，可以透過這個方法完全禁止全域物件或將其限制在安全的子集合。

以下是一個僅允許從*builtins* 模組中載入少數安全類🄵的拆封器（unpickler）的例子：：

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # 只允許幾個🄵建的安全類🄵
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # 完全禁止任何其他類🄵
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """一個模擬 pickle.loads() 的輔助函式"""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

我們剛才實作的的拆封器範例正常運作的樣子：：

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

正如我們的範例所示，必須謹慎審視能被拆封的🄵容。因此，如果你的應用場景非常關心安全性，你可能需要考慮其他選擇，例如*xmlrpc.client* 中的 marshalling API 或其他第三方解🄵方案。

12.1.9 效能

較近期的 pickle 協定版本（從 2 版協定開始）🄵多種常見功能和🄵建型🄵提供了高效率的二進位編碼。此外，*pickle* 模組還具備一個透明化的、以 C 語言編寫的最佳化工具。

12.1.10 范例

最簡單的使用方式，呼叫*dump()* 和*load()* 函式。：

```
import pickle

# 任意 pickle 支援的物件。
```

(繼續下一頁)

(繼續上一頁)

```
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # 使用可用的最高協定來封裝 'data' 字典。
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

以下範例可以讀取前述程式所封裝的 pickle 資料。:

```
import pickle

with open('data.pickle', 'rb') as f:
    # 會自動檢測資料使用的協定版本，因此我們不需要手動指定。
    data = pickle.load(f)
```

12.1.11 命令列介面

The *pickle* module can be invoked as a script from the command line, it will display contents of the pickle files. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode, see *pickletools CLI usage*.

```
python -m pickle pickle_file [pickle_file ...]
```

接受以下選項:

pickle_file

A pickle file to read, or - to indicate reading from standard input.

也參考

***copyreg* 模組**
🄵🄵擴充型🄵的 Pickle 介面建構子。

***pickletools* 模組**
用於分析或處🄵被封裝資料的工具。

***shelve* 模組**
索引式資料庫；使用*pickle* 實作。

***copy* 模組**
物件的淺層或深度拷貝。

***marshal* 模組**
🄵建型🄵的高效能序列化。

🄵解

12.2 copyreg --- 🄵🄵 pickle 支援函式

原始碼： Lib/copyreg.py

---

copyreg 模組提供了一種用以定義在 pickle 特定物件時使用之函式的方式。*pickle* 和*copy* 模組在 pickle/copy 這些物件時使用這些函式。此模組提供有關非類🄵物件之建構函式的配置資訊。此類建構函式可以是工廠函式 (factory function) 或類🄵實例。

copyreg.**constructor**(*object*)

宣告 *object* 是一個有效的建構函式。如果 *object* 不可呼叫（因此不可作🄵有效的建構函式），則會引發 *TypeError*。

copyreg.**pickle**(*type*, *function*, *constructor_ob=None*)

宣告 *function* 應該用作 *type* 型🄵之物件的「歸約 (”reduction”)」函式。*function* 必須回傳字串或包含 2 到 6 個元素的元組。有關 *function* 介面的更多詳細資訊，請參🄵*dispatch_table*。

constructor_ob 參數是一個遺留功能，現在已被忽略，但如果要傳遞它的話則必須是個可呼叫物件。

請注意，pickler 物件或*pickle.Pickler* 子類🄵的*dispatch_table* 屬性也可用於宣告歸約函式。

12.2.1 范例

下面範例展示如何🄵🄵一個 pickle 函式以及如何使用它：

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 `shelve` --- Python object persistence

原始碼： Lib/shelve.py

A ”shelf” is a persistent, dictionary-like object. The difference with ”dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects --- anything that the *pickle* module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional *flag* parameter has the same interpretation as the *flag* parameter of *dbm.open()*.

By default, pickles created with *pickle.DEFAULT_PROTOCOL* are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see 範例). If the optional *writeback* parameter is set to `True`, all entries accessed are also cached in memory, and written back on *sync()* and *close()*; this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

在 3.10 版的變更: *pickle.DEFAULT_PROTOCOL* is now used as the default pickle protocol.

在 3.11 版的變更: Accepts *path-like object* for filename.

備🄵

Do not rely on the shelf being closed automatically; always call `close()` explicitly when you don't need it any more, or use `shelve.open()` as a context manager:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

警告

Because the `shelve` module is backed by `pickle`, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support most of methods and operations supported by dictionaries (except copying, constructors and operators `|` and `|=`). This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

`Shelf.`**`sync`**`()`

Write back all entries in the cache if the shelf was opened with *writeback* set to `True`. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with `close()`.

`Shelf.`**`close`**`()`

Synchronize and close the persistent *dict* object. Operations on a closed shelf will fail with a `ValueError`.

也參考

Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

12.3.1 Restrictions

- The choice of which database package will be used (such as `dbm.ndbm` or `dbm.gnu`) depends on which interface is available. Therefore it is not safe to open the database directly using `dbm`. The database is also (unfortunately) subject to the limitations of `dbm`, if it is used --- this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- The `shelve` module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.
- On macOS `dbm.ndbm` can silently corrupt the database file on updates, which can cause hard crashes when trying to read from the database.

`class` `shelve.`**`Shelf`**`(`*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*`)`

A subclass of `collections.abc.MutableMapping` which stores pickled values in the *dict* object.

By default, pickles created with `pickle.DEFAULT_PROTOCOL` are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the `pickle` documentation for a discussion of the pickle protocols.

If the *writeback* parameter is `True`, the object will hold a cache of all entries accessed and write them back to the *dict* at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

The *keyencoding* parameter is the encoding used to encode keys before they are used with the underlying dict.

A *Shelf* object can also be used as a context manager, in which case it will be automatically closed when the `with` block ends.

在 3.2 版的變更: Added the *keyencoding* parameter; previously, keys were always encoded in UTF-8.

在 3.4 版的變更: 新增情境管理器的支援。

在 3.10 版的變更: *pickle.DEFAULT_PROTOCOL* is now used as the default pickle protocol.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of *Shelf* which exposes `first()`, `next()`, `previous()`, `last()` and `set_location()` methods. These are available in the third-party `bsddb` module from pybsddb but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of `bsddb.hashopen()`, `bsddb.btopen()` or `bsddb.rnopen()`. The optional *protocol*, *writeback*, and *keyencoding* parameters have the same interpretation as for the *Shelf* class.

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

A subclass of *Shelf* which accepts a *filename* instead of a dict-like object. The underlying file will be opened using *dbm.open()*. By default, the file will be created and opened for both read and write. The optional *flag* parameter has the same interpretation as for the *open()* function. The optional *protocol* and *writeback* parameters have the same interpretation as for the *Shelf* class.

12.3.2 范例

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

也參考

dbm 模組
Generic interface to `dbm`-style databases.

pickle 模組
Object serialization used by shelve.

12.4 marshal --- F部 Python 物件序列化

此模組包含一個能以二進位制格式來讀寫 Python 值的函式。這種格式是 Python 專屬但獨立於機器架構的（例如，你可以在一臺 PC 上寫入某個 Python 值，再將檔案傳到一臺 Mac 上F在那F讀取它）。這種格式的細節是有意地不在文件上F明的；它可能在不同 Python 版本中被改變（雖然這種情F極少發生）。[1]

這不是一個通用「持久性 (persistence)」模組。關於通用持久性以及透過 RPC 呼叫傳遞 Python 物件，請參F*pickle* 和*shelve* 等模組。*marshal* 模組主要是F了支援用來讀寫 `.pyc` 檔案之 Python 模組的「F編譯 (pseudo-compiled)」程式碼。因此，Python 維護者保留了在必要時以不向後相容的方式修改 marshal 格式的權利。程式碼物件的格式在不同 Python 版本間是不相容的，即使格式版本相同。在不正確的 Python 版本中反序列化程式碼物件會有未定義的行F。如果你要序列化和反序列化 Python 物件，請改用*pickle* 模組 -- 其執行效率相當、有保證版本獨立性，且實質上 pickle 還支援比 marshal 更多樣的物件。

警告

marshal 模組對於錯誤或惡意構建的資料來F是不安全的。永遠不要 unmarshal 來自不受信任的或來源未經驗證的資料。

有些函式可以讀/寫檔案，還有些函式可以操作類位元組串物件 (bytes-like object)。

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported:

- Numeric types: *int*, *bool*, *float*, *complex*.
- Strings (*str*) and *bytes*. *Bytes-like objects* like *bytearray* are marshalled as `bytes`.
- Containers: *tuple*, *list*, *set*, *frozenset*, and (since *version* 5), *slice*. It should be understood that these are supported only if the values contained therein are themselves supported. Recursive containers are supported since *version* 3.
- The singletons *None*, *Ellipsis* and *StopIteration*.
- *code* objects, if *allow_code* is true. See note above about version dependence.

在 3.4 版的變更:

- Added format version 3, which supports marshalling recursive lists, sets and dictionaries.
- Added format version 4, which supports efficient representations of short strings.

在 3.14 版的變更: Added format version 5, which allows marshalling slices.

這個模組定義了以下函式:

`marshal.`**dump**(*value*, *file*, *version=version*, */*, ***, *allow_code=True*)

將值寫入被開F的檔案。值必須F受支援的型F，檔案必須F可寫入的*binary file*。

如果值具有（或其所包含的物件具有）不支援的型F，則會引發*ValueError* 例外 --- 但是垃圾資料 (garbage data) 也將寫入檔案，物件也無法正確地透過*load()* 重新讀取。程式碼物件只有在 *allow_code* F true 時才會支援。

version 引數指明 `dump` 應該使用的資料格式（見下文）。

引發一個附帶引數 `value` 與 `version` 的稽核事件 *(auditing event)* `marshal.dumps`。

[1] 此模組的名稱來源於 Modula-3 (及其他語言) 的設計者所使用的術語，他們使用"marshal" 來表示自包含 (self-contained) 形式資料的傳輸。嚴格來F，將資料從F部形式轉FF外部形式 (例如用於 RPC 緩衝區) 稱F"marshal"，而其反向過程則稱F"unmarshal"。

在 3.13 版的變更: 新增 *allow_code* 參數。

marshal.**load**(*file*, */*, *, *allow_code=True*)

從開🄵的檔案讀取一個值🄵回傳。如果讀不到有效的值（例如，由於資料🄵不同 Python 版本的不相容 marshal 格式），則會引發*EOFError*、*ValueError* 或*TypeError*。程式碼物件只有在 *allow_code* 🄵 true 時才會支援。檔案必須🄵可讀取的*binary file*。

引發一個🄵有附帶引數的稽核事件 marshal.load。

備🄵

如果透過*dump()* marshal 了一個包含不支援型🄵的物件，*load()* 會將不可 marshal 的型🄵替🄵🄵 None。

在 3.10 版的變更: 使用此呼叫🄵每個程式碼物件引發一個 code.__new__ 稽核事件。現在它會🄵整個載入操作引發單個 marshal.load 事件。

在 3.13 版的變更: 新增 *allow_code* 參數。

marshal.**dumps**(*value*, *version=version*, */*, *, *allow_code=True*)

回傳將透過 dump(value, file) 來被寫入一個檔案的位元組串物件，其值必須是有支援的型🄵，如果值（或其包含的任一物件）🄵不支援的型🄵則會引發*ValueError*。程式碼物件只有在 *allow_code* 🄵 true 時才會支援。

version 引數指明 dumps 應當使用的資料型🄵（見下文）。

引發一個附帶引數 value 與 version 的稽核事件 *(auditing event)* marshal.dumps。

在 3.13 版的變更: 新增 *allow_code* 參數。

marshal.**loads**(*bytes*, */*, *, *allow_code=True*)

將*bytes-like object* 轉🄵🄵一個值。如果找不到有效的值，則會引發*EOFError*、*ValueError* 或*TypeError*。程式碼物件只有在 *allow_code* 🄵 true 時才會支援。輸入中額外的位元組串會被忽略。

引發一個附帶引數 bytes 的稽核事件 marshal.loads。

在 3.10 版的變更: 使用此呼叫🄵每個程式碼物件引發一個 code.__new__ 稽核事件。現在它會🄵整個載入操作引發單個 marshal.loads 事件。

在 3.13 版的變更: 新增 *allow_code* 參數。

此外，還定義了以下常數:

marshal.**version**

表示模組所使用的格式。第 0 版🄵歷史上第一版本；後續版本會新增功能。當引入新版本時，它通常會成🄵預設版本。

版本	Available since	New features
1	Python 2.4	共享駐留字串
2	Python 2.5	Binary representation of floats
3	Python 3.4	Support for object instancing and recursion
4	Python 3.4	Efficient representation of short strings
5	Python 3.14	Support for *slice* objects

🄵解

12.5 dbm --- Unix "databases" 的介面

原始碼： Lib/dbm/__init__.py

dbm is a generic interface to variants of the DBM database:

- *dbm.sqlite3*
- *dbm.gnu*
- *dbm.ndbm*

If none of these modules are installed, the slow-but-simple implementation in module *dbm.dumb* will be used. There is a third party interface to the Oracle Berkeley DB.

exception dbm.**error**

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named *dbm.error* as the first item --- the latter is used when *dbm.error* is raised.

dbm.**whichdb**(*filename*)

This function attempts to guess which of the several simple database modules available --- *dbm.sqlite3*, *dbm.gnu*, *dbm.ndbm*, or *dbm.dumb* --- should be used to open a given file.

回傳以下其中一個值:

- `None` if the file can't be opened because it's unreadable or doesn't exist
- the empty string (`''`) if the file's format can't be guessed
- a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`

在 3.11 版的變更: *filename* accepts a *path-like object*.

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)

Open a database and return the corresponding database object.

參數

- **file** (*path-like object*) -- 要打開的資料庫檔案

  If the database file already exists, the *whichdb()* function is used to determine its type and the appropriate module is used; if it does not exist, the first submodule listed above that can be imported is used.

- **flag** (*str*) --
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** (*int*) -- The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

在 3.11 版的變更: *file* 接受一個類路徑物件。

The object returned by *open()* supports the same basic functionality as a *dict*; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()` methods.

Key and values are always stored as *bytes*. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

在 3.2 版的變更: `get()` and `setdefault()` methods are now available for all *dbm* backends.

在 3.4 版的變更: Added native support for the context management protocol to the objects returned by *open()*.

在 3.8 版的變更: Deleting a key from a read-only database raises a database module specific exception instead of *KeyError*.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

也參考

***shelve* 模組**
Persistence module which stores non-string data.

The individual submodules are described in the following sections.

12.5.1 `dbm.sqlite3` --- dbm 的 SQLite 後端

在 3.13 版被加入.

原始碼： Lib/dbm/sqlite3.py

This module uses the standard library *sqlite3* module to provide an SQLite backend for the *dbm* module. The files created by *dbm.sqlite3* can thus be opened by *sqlite3*, or any other SQLite browser, including the SQLite CLI.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

`dbm.sqlite3.`**`open`**(*filename*, */*, *flag='r'*, *mode=0o666*)

Open an SQLite database. The returned object behaves like a *mapping*, implements a `close()` method, and supports a ”closing” context manager via the `with` keyword.

參數

- **`filename`** (*path-like object*) -- 要打開的資料庫路徑
- **`flag`** (`str`) --
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.

  - `'n'`: Always create a new, empty database, open for reading and writing.

- **mode** -- The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

12.5.2 `dbm.gnu` --- GNU 資料庫管理器

原始碼： Lib/dbm/gnu.py

The *dbm.gnu* module provides an interface to the GDBM (GNU dbm) library, similar to the *dbm.ndbm* module, but with additional functionality like crash tolerance.

備🄵

The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

exception dbm.gnu.**error**

Raised on *dbm.gnu*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.gnu.**open**(*filename*, *flag='r'*, *mode=0o666*, /)

打開一個 GDBM 資料庫，🄵回傳一個 `gdbm` 物件。

參數

- **filename** (*path-like object*) -- 要打開的資料庫檔案
- **flag** (`str`) --
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.

  The following additional characters may be appended to control how the database is opened:

  - `'f'`: Open the database in fast mode. Writes to the database will not be synchronized.
  - `'s'`: Synchronized mode. Changes to the database will be written immediately to the file.
  - `'u'`: 不要鎖住資料庫。

  Not all flags are valid for all versions of GDBM. See the *open_flags* member for a list of supported flag characters.
- **mode** (`int`) -- The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

引發

error -- 如果一個無效的 *flag* 引數被傳入。

在 3.11 版的變更: *filename* accepts a *path-like object*.

dbm.gnu.**open_flags**

A string of characters the *flag* parameter of *open()* supports.

gdbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided:

gdbm.**firstkey**()

It's possible to loop over every key in the database using this method and the *nextkey()* method. The traversal is ordered by GDBM's internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.**nextkey**(*key*)

Returns the key that follows *key* in the traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()

If you have carried out a lot of deletions and would like to shrink the space used by the GDBM file, this routine will reorganize the database. gdbm objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.**sync**()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.**close**()

關閉 GDBM 資料庫。

gdbm.**clear**()

移除 GDBM 資料庫的所有項目。

在 3.13 版被加入.

12.5.3 dbm.ndbm --- 新資料庫管理器

原始碼： Lib/dbm/ndbm.py

The *dbm.ndbm* module provides an interface to the NDBM (New Database Manager) library. This module can be used with the "classic" NDBM interface or the GDBM compatibility interface.

備🄵

The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

警告

The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

exception dbm.ndbm.**error**

Raised on *dbm.ndbm*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.ndbm.**library**

Name of the NDBM implementation library used.

dbm.ndbm.**open**(*filename*, *flag='r'*, *mode=0o666*, */*)

Open an NDBM database and return an ndbm object.

參數

- **filename** (*path-like object*) -- The basename of the database file (without the .dir or .pag extensions).
- **flag** (str) --
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

ndbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided:

在 3.11 版的變更: Accepts *path-like object* for filename.

ndbm.**close**()

關閉 NDBM 資料庫。

ndbm.**clear**()

移除 NDBM 資料庫的所有項目。

在 3.13 版被加入.

12.5.4 dbm.dumb --- 可🄵式 DBM 實作

原始碼： Lib/dbm/dumb.py

> **備🄵**
>
> The *dbm.dumb* module is intended as a last resort fallback for the *dbm* module when a more robust module is not available. The *dbm.dumb* module is not written for speed and is not nearly as heavily used as the other database modules.

The *dbm.dumb* module provides a persistent *dict*-like interface which is written entirely in Python. Unlike other *dbm* backends, such as *dbm.gnu*, no external library is required.

dbm.dumb 模組定義了以下項目：

exception dbm.dumb.**error**

Raised on *dbm.dumb*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.dumb.**open**(*filename*, *flag='c'*, *mode=0o666*)

Open a dbm.dumb database. The returned database object behaves similar to a *mapping*, in addition to providing *sync()* and *close()* methods.

參數

- **filename** -- The basename of the database file (without extensions). A new database creates the following files:
  - *filename*.dat
  - *filename*.dir
- **flag** (str) --
  - 'r': Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c' (default): Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

> **⚠ 警告**
>
> It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python's AST compiler.

在 3.5 版的變更: *open()* always creates a new database when *flag* is 'n'.

在 3.8 版的變更: A database opened read-only if *flag* is 'r'. A database is not created if it does not exist if *flag* is 'r' or 'w'.

在 3.11 版的變更: *filename* accepts a *path-like object*.

In addition to the methods provided by the *collections.abc.MutableMapping* class, the following methods are provided:

dumbdbm.**sync**()

Synchronize the on-disk directory and data files. This method is called by the *shelve.Shelf.sync()* method.

dumbdbm.**close**()

關閉資料庫。

12.6 sqlite3 --- SQLite 資料庫的 DB-API 2.0 介面

原始碼: Lib/sqlite3/ SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.15.2 or newer.

此文件包含四個主要章節:

- *教學* 教導如何使用 sqlite3 模組。
- *Reference* 描述此模組定義的類🄴與函式。
- *How-to guides* 詳細🄴明如何處理特定工作。

- 解釋 深入提供交易 (transaction) 控制的背景。

也參考

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

https://www.w3schools.com/sql/
Tutorial, reference and examples for learning SQL syntax.

PEP 249 - 資料庫 API 規格 2.0
PEP 由 Marc-André Lemburg 撰寫。

12.6.1 教學

In this tutorial, you will create a database of Monty Python movies using basic `sqlite3` functionality. It assumes a fundamental understanding of database concepts, including cursors and transactions.

First, we need to create a new database and open a database connection to allow `sqlite3` to work with it. Call *`sqlite3.connect()`* to create a connection to the database `tutorial.db` in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned *`Connection`* object `con` represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call *`con.cursor()`* to create the *`Cursor`*:

```
cur = con.cursor()
```

Now that we've got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration -- thanks to the flexible typing feature of SQLite, specifying the data types is optional. Execute the `CREATE TABLE` statement by calling *`cur.execute(...)`*:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table for details). Execute that query by calling *`cur.execute(...)`*, assign the result to `res`, and call *`res.fetchone()`* to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a *`tuple`* containing the table's name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling *`cur.execute(...)`*:

```
cur.execute("""
    INSERT INTO movie VALUES
```

(繼續下一頁)

(繼續上一頁)

```
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see *Transaction control* for details). Call *con.commit()* on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar *cur.execute(...)* to assign the result to `res`, and call *res.fetchall()* to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a *list* of two `tuple`s, one per row, each containing that row's `score` value.

Now, insert three more rows by calling *cur.executemany(...)*:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks (see *How to use placeholders to bind values in SQL queries* for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item *tuple* of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling *con.close()* to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail', released in 1975
>>> new_con.close()
```

You've now created an SQLite database using the `sqlite3` module, inserted data and retrieved values from it in multiple ways.

也參考

- 進一步參考*How-to guides*:
  - *How to use placeholders to bind values in SQL queries*
  - *How to adapt custom Python types to SQLite values*
  - *How to convert SQLite values to custom Python types*
  - *How to use the connection context manager*
  - *How to create and use row factories*
- 解釋 for in-depth background on transaction control.

12.6.2 Reference

Module functions

sqlite3.**connect**(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

Open a connection to an SQLite database.

參數

- **database** (*path-like object*) -- The path to the database file to be opened. You can pass `":memory:"` to create an SQLite database existing only in memory, and open a connection to it.
- **timeout** (`float`) -- How many seconds the connection should wait before raising an `OperationalError` when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.
- **detect_types** (`int`) -- Control whether and how data types not *natively supported by SQLite* are looked up to be converted to Python types, using the converters registered with `register_converter()`. Set it to any combination (using `|`, bitwise or) of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to enable this. Column names takes precedence over declared types if both flags are set. By default (`0`), type detection is disabled.
- **isolation_level** (`str | None`) -- Control legacy transaction handling behaviour. See `Connection.isolation_level` and *Transaction control via the isolation_level attribute* for more information. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"`; or `None` to disable opening transactions implicitly. Has no effect unless `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).
- **check_same_thread** (`bool`) -- If `True` (default), `ProgrammingError` will be raised if the database connection is used by a thread other than the one that created it. If `False`, the connection may be accessed in multiple threads; write operations may need to be serialized by the user to avoid data corruption. See `threadsafety` for more information.
- **factory** (`Connection`) -- A custom subclass of `Connection` to create the connection with, if not the default `Connection` class.
- **cached_statements** (`int`) -- The number of statements that `sqlite3` should internally cache for this connection, to avoid parsing overhead. By default, 128 statements.
- **uri** (`bool`) -- If set to `True`, *database* is interpreted as a URI (Uniform Resource Identifier) with a file path and an optional query string. The scheme part *must* be `"file:"`, and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various *How to work with SQLite URIs*.

- **autocommit** (bool) -- Control **PEP 249** transaction handling behaviour. See *Connection.autocommit* and *Transaction control via the autocommit attribute* for more information. *autocommit* currently defaults to *LEGACY_TRANSACTION_CONTROL*. The default will change to False in a future Python release.

回傳型🄵
Connection

引發一個附帶引數 database 的稽核事件 sqlite3.connect。

引發一個附帶引數 connection_handle 的稽核事件 sqlite3.connect/handle。

在 3.4 版的變更: 新增 *uri* 參數。

在 3.7 版的變更: *database* can now also be a *path-like object*, not only a string.

在 3.10 版的變更: 新增 sqlite3.connect/handle 稽核事件。

在 3.12 版的變更: 新增 *autocommit* 參數。

在 3.13 版的變更: Positional use of the parameters *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, and *uri* is deprecated. They will become keyword-only parameters in Python 3.15.

sqlite3.**complete_statement**(*statement*)

Return True if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

範例:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling *execute()*.

See runsource() in Lib/sqlite3/__main__.py for real-world use.

sqlite3.**enable_callback_tracebacks**(*flag*, /)

Enable or disable callback tracebacks. By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to True. Afterwards, you will get tracebacks from callbacks on *sys.stderr*. Use False to disable the feature again.

> **備🄵**
>
> Errors in user-defined function callbacks are logged as unraisable exceptions. Use an *unraisable hook handler* for introspection of the failed callback.

sqlite3.**register_adapter**(*type*, *adapter*, /)

Register an *adapter* *callable* to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a *type that SQLite natively understands*.

sqlite3.**register_converter**(*typename*, *converter*, /)

Register the *converter* *callable* to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a *bytes* object and should return an object of the desired Python type. Consult the parameter *detect_types* of *connect()* for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

sqlite3.**LEGACY_TRANSACTION_CONTROL**

Set *autocommit* to this constant to select old style (pre-Python 3.12) transaction control behaviour. See *Transaction control via the isolation_level attribute* for more information.

sqlite3.**PARSE_DECLTYPES**

Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function using the declared types for each column. The types are declared when the database table is created. `sqlite3` will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named "integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
)
```

This flag may be combined with *PARSE_COLNAMES* using the `|` (bitwise or) operator.

> **備🄵**
>
> Generated fields (for example `MAX(p)`) are returned as *str*. Use `PARSE_COLNAMES` to enforce types for such queries.

sqlite3.**PARSE_COLNAMES**

Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The query column name must be wrapped in double quotes (`"`) and the type name must be wrapped in square brackets (`[]`).

```
SELECT MAX(p) as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with *PARSE_DECLTYPES* using the `|` (bitwise or) operator.

sqlite3.**SQLITE_OK**

sqlite3.**SQLITE_DENY**

sqlite3.**SQLITE_IGNORE**

Flags that should be returned by the *authorizer_callback* *callable* passed to *Connection.set_authorizer()*, to indicate whether:

- Access is allowed (`SQLITE_OK`),
- The SQL statement should be aborted with an error (`SQLITE_DENY`)
- The column should be treated as a `NULL` value (`SQLITE_IGNORE`)

sqlite3.**apilevel**

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to `"2.0"`.

sqlite3.**paramstyle**

String constant stating the type of parameter marker formatting expected by the `sqlite3` module. Required by the DB-API. Hard-coded to `"qmark"`.

> **備🄵**
>
> The `named` DB-API parameter style is also supported.

sqlite3.**sqlite_version**

Version number of the runtime SQLite library as a *string*.

sqlite3.**sqlite_version_info**

Version number of the runtime SQLite library as a *tuple* of *integers*.

sqlite3.**threadsafety**

Integer constant required by the DB-API 2.0, stating the level of thread safety the sqlite3 module supports. This attribute is set based on the default threading mode the underlying SQLite library is compiled with. The SQLite threading modes are:

1. **Single-thread**: In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread**: In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**: In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows:

SQLite 執行緒模式	執行緒安全	SQLITE_THREADSA	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

在 3.11 版的變更: Set *threadsafety* dynamically instead of hard-coding it to 1.

sqlite3.**SQLITE_DBCONFIG_DEFENSIVE**

sqlite3.**SQLITE_DBCONFIG_DQS_DDL**

sqlite3.**SQLITE_DBCONFIG_DQS_DML**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FKEY**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER**

sqlite3.**SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION**

sqlite3.**SQLITE_DBCONFIG_ENABLE_QPSG**

sqlite3.**SQLITE_DBCONFIG_ENABLE_TRIGGER**

sqlite3.**SQLITE_DBCONFIG_ENABLE_VIEW**

sqlite3.**SQLITE_DBCONFIG_LEGACY_ALTER_TABLE**

sqlite3.**SQLITE_DBCONFIG_LEGACY_FILE_FORMAT**

sqlite3.**SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE**

sqlite3.**SQLITE_DBCONFIG_RESET_DATABASE**

sqlite3.**SQLITE_DBCONFIG_TRIGGER_EQP**

sqlite3.**SQLITE_DBCONFIG_TRUSTED_SCHEMA**

sqlite3.**SQLITE_DBCONFIG_WRITABLE_SCHEMA**

These constants are used for the *Connection.setconfig()* and *getconfig()* methods.

The availability of these constants varies depending on the version of SQLite Python was compiled with.

在 3.12 版被加入.

也參考

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html
SQLite docs: Database Connection Configuration Options

Deprecated since version 3.12, removed in version 3.14: The version and version_info constants.

Connection 物件

class sqlite3.**Connection**

Each open SQLite database is represented by a `Connection` object, which is created using *sqlite3.connect()*. Their main purpose is creating *Cursor* objects, and *Transaction control*.

> **也參考**
>
> - *How to use connection shortcut methods*
> - *How to use the connection context manager*

在 3.13 版的變更: A *ResourceWarning* is emitted if *close()* is not called before a `Connection` object is deleted.

一個 SQLite 資料庫連🄵具有以下屬性和方法:

cursor(*factory=Cursor*)

Create and return a *Cursor* object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a *callable* returning an instance of *Cursor* or its subclasses.

blobopen(*table*, *column*, *row*, */*, ***, *readonly=False*, *name='main'*)

Open a *Blob* handle to an existing BLOB (Binary Large OBject).

參數

- **table** (str) -- The name of the table where the blob is located.
- **column** (str) -- The name of the column where the blob is located.
- **row** (str) -- The name of the row where the blob is located.
- **readonly** (bool) -- Set to `True` if the blob should be opened without write permissions. Defaults to `False`.
- **name** (str) -- The name of the database where the blob is located. Defaults to `"main"`.

引發

OperationalError -- When trying to open a blob in a `WITHOUT ROWID` table.

回傳型🄵

Blob

> **備🄵**
>
> The blob size cannot be changed using the *Blob* class. Use the SQL function `zeroblob` to create a blob with a fixed size.

在 3.11 版被加入.

commit()

Commit any pending transaction to the database. If *autocommit* is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was committed by this method.

rollback()

Roll back to the start of any pending transaction. If *autocommit* is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If *autocommit* is `False`, any pending transaction is implicitly rolled back. If `autocommit` is `True` or *LEGACY_TRANSACTION_CONTROL*, no implicit transaction control is executed. Make sure to *commit()* before closing to avoid losing pending changes.

execute(*sql*, *parameters=()*, */*)

Create a new *Cursor* object and call *execute()* on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*, *parameters*, */*)

Create a new *Cursor* object and call *executemany()* on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*, */*)

Create a new *Cursor* object and call *executescript()* on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *narg*, *func*, *, *deterministic=False*)

Create or remove a user-defined SQL function.

參數

- **name** (`str`) -- The name of the SQL function.
- **narg** (`int`) -- The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.
- **func** (*callback* | None) -- A *callable* that is called when the SQL function is invoked. The callable must return *a type natively supported by SQLite*. Set to `None` to remove an existing SQL function.
- **deterministic** (`bool`) -- If `True`, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

在 3.8 版的變更: 新增 *deterministic* 參數。

範例:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

在 3.13 版的變更: Passing *name*, *narg*, and *func* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

參數

- **name** (`str`) -- The name of the SQL aggregate function.
- **n_arg** (`int`) -- The number of arguments the SQL aggregate function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (*class* | None) -- A class must implement the following methods:
  - `step()`: Add a row to the aggregate.
  - `finalize()`: Return the final result of the aggregate as *a type natively supported by SQLite*.

The number of arguments that the `step()` method must accept is controlled by *n_arg*.

Set to `None` to remove an existing SQL aggregate function.

範例：

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

在 3.13 版的變更: Passing *name*, *n_arg*, and *aggregate_class* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_window_function(*name*, *num_params*, *aggregate_class*, */*)

Create or remove a user-defined aggregate window function.

參數

- **name** (`str`) -- The name of the SQL aggregate window function to create or remove.
- **num_params** (`int`) -- The number of arguments the SQL aggregate window function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (*class* | None) -- A class that must implement the following methods:
  - `step()`: Add a row to the current window.
  - `value()`: Return the current value of the aggregate.
  - `inverse()`: Remove a row from the current window.
  - `finalize()`: Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the `step()` and `value()` methods must accept is controlled by *num_params*.

  Set to `None` to remove an existing SQL aggregate window function.

引發

NotSupportedError -- If used with a version of SQLite older than 3.25.0, which does not support aggregate window functions.

在 3.11 版被加入.

範例：

```
# Example taken from https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
```

(繼續下一頁)

(繼續上一頁)

```
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, */*)

Create a collation named *name* using the collating function *callable*. *callable* is passed two *string* arguments, and it should return an *integer*:

- `1` if the first is ordered higher than the second
- `-1` if the first is ordered lower than the second
- `0` if they are ordered equal

The following example shows a reverse sorting collation:

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1
```

(繼續下一頁)

(繼續上一頁)

```
con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",), ("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

Remove a collation function by setting *callable* to `None`.

在 3.11 版的變更: The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

interrupt()

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an *OperationalError*.

set_authorizer(*authorizer_callback*)

Register *callable* *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of *SQLITE_OK*, *SQLITE_DENY*, or *SQLITE_IGNORE* to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

Passing `None` as *authorizer_callback* will disable the authorizer.

在 3.11 版的變更: Added support for disabling the authorizer using `None`.

在 3.13 版的變更: Passing *authorizer_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_progress_handler(*progress_handler*, *n*)

Register *callable* *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a *DatabaseError* exception.

在 3.13 版的變更: Passing *progress_handler* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_trace_callback(*trace_callback*)

Register *callable* *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as *str*) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the *transaction management* of the `sqlite3` module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

> **備🄵**
>
> Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use *enable_callback_tracebacks()* to enable printing tracebacks from exceptions raised in the trace callback.

在 3.3 版被加入.

在 3.13 版的變更: Passing *trace_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

enable_load_extension(*enabled*, /)

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

> **備🄵**
>
> The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the `--enable-loadable-sqlite-extensions` option to **configure**.

引發一個附帶引數 `connection`、`enabled` 的稽核事件 `sqlite3.enable_load_extension`。

在 3.2 版被加入.

在 3.10 版的變更: 加入 `sqlite3.enable_load_extension` 稽核事件。

```
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin onions␣
↪garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie', 'broccoli cheese␣
↪onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin sugar␣
↪flour butter');
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE name␣
↪MATCH 'pie'"):
    print(row)
```

load_extension(*path*, /, *, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with *enable_load_extension()* before calling this method.

參數

- **path** (str) -- The path to the SQLite extension.
- **entrypoint** (str | *None*) -- Entry point name. If None (the default), SQLite will come up with an entry point name of its own; see the SQLite docs Loading an Extension for details.

引發一個附帶引數 connection、path 的稽核事件 sqlite3.load_extension。

在 3.2 版被加入.

在 3.10 版的變更: 加入 sqlite3.load_extension 稽核事件。

在 3.12 版的變更: 新增 *entrypoint* 參數。

iterdump(*, *filter=None*)

Return an *iterator* to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the .dump command in the **sqlite3** shell.

參數

filter (str | *None*) -- An optional LIKE pattern for database objects to dump, e.g. prefix_%. If None (the default), all database objects will be included.

範例:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

也參考

如何處理非 UTF-8 的文字編碼

在 3.13 版的變更: 新增 *filter* 參數。

backup(*target*, *, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

參數

- **target** (Connection) -- The database connection to save the backup to.
- **pages** (int) -- The number of pages to copy at a time. If equal to or less than 0, the entire database is copied in a single step. Defaults to -1.
- **progress** (*callback* | None) -- If set to a *callable*, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to None.
- **name** (str) -- The name of the database to back up. Either "main" (the default) for the main database, "temp" for the temporary database, or the name of a custom database as attached using the ATTACH DATABASE SQL statement.
- **sleep** (float) -- The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

在 3.7 版被加入.

也參考

如何處理非 UTF-8 的文字編碼

getlimit(*category*, */*)

Get a connection runtime limit.

參數

category (int) -- The SQLite limit category to be queried.

回傳型🄵

int

引發

ProgrammingError -- If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for *Connection* `con` (the default is 1000000000):

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

在 3.11 版被加入.

setlimit(*category*, *limit*, */*)

Set a connection runtime limit. Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

參數

- **category** (int) -- The SQLite limit category to be set.
- **limit** (int) -- The value of the new limit. If negative, the current limit is unchanged.

回傳型🄵

int

引發

ProgrammingError -- If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for *Connection* `con` (the default limit is 10):

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

在 3.11 版被加入.

getconfig(*op*, */*)

Query a boolean connection configuration option.

參數

op (int) -- A *SQLITE_DBCONFIG code*.

回傳型🄵

bool

在 3.12 版被加入.

setconfig(*op*, *enable=True*, */*)

Set a boolean connection configuration option.

參數

- **op** (int) -- A *SQLITE_DBCONFIG code*.
- **enable** (bool) -- `True` if the configuration option should be enabled (default); `False` if it should be disabled.

在 3.12 版被加入.

serialize(*, *name='main'*)

Serialize a database into a *bytes* object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

參數

name (str) -- The database name to be serialized. Defaults to `"main"`.

回傳型🄵

bytes

> **備🄵**
>
> This method is only available if the underlying SQLite library has the serialize API.

在 3.11 版被加入.

deserialize(*data*, */*, *, *name='main'*)

Deserialize a *serialized* database into a *Connection*. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serialization contained in *data*.

參數

- **data** (bytes) -- A serialized database.
- **name** (str) -- The database name to deserialize into. Defaults to `"main"`.

引發

- ***OperationalError*** -- If the database connection is currently involved in a read transaction or a backup operation.
- ***DatabaseError*** -- If *data* does not contain a valid SQLite database.
- ***OverflowError*** -- If *len(data)* is larger than `2**63 - 1`.

> **備🄵**
>
> This method is only available if the underlying SQLite library has the deserialize API.

在 3.11 版被加入.

autocommit

This attribute controls **PEP 249**-compliant transaction behaviour. `autocommit` has three allowed values:

- `False`: Select **PEP 249**-compliant transaction behaviour, implying that `sqlite3` ensures a transaction is always open. Use *`commit()`* and *`rollback()`* to close transactions.

  This is the recommended value of `autocommit`.

- `True`: Use SQLite's autocommit mode. *`commit()`* and *`rollback()`* have no effect in this mode.

- *`LEGACY_TRANSACTION_CONTROL`*: Pre-Python 3.12 (non-**PEP 249**-compliant) transaction control. See *`isolation_level`* for more details.

  This is currently the default value of `autocommit`.

Changing `autocommit` to `False` will open a new transaction, and changing it to `True` will commit any pending transaction.

更多詳情請見*Transaction control via the autocommit attribute*。

> **備🄵**
>
> The *`isolation_level`* attribute has no effect unless *`autocommit`* is *`LEGACY_TRANSACTION_CONTROL`*.

在 3.12 版被加入.

in_transaction

This read-only attribute corresponds to the low-level SQLite autocommit mode.

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

在 3.2 版被加入.

isolation_level

Controls the *legacy transaction handling mode* of `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, *implicit transaction management* is performed.

If not overridden by the *isolation_level* parameter of *`connect()`*, the default is `""`, which is an alias for `"DEFERRED"`.

> **備🄵**
>
> Using *`autocommit`* to control transaction handling is recommended over using `isolation_level`. `isolation_level` has no effect unless *`autocommit`* is set to *`LEGACY_TRANSACTION_CONTROL`* (the default).

row_factory

The initial *`row_factory`* for *`Cursor`* objects created from this connection. Assigning to this attribute does not affect the `row_factory` of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a *`tuple`*.

更多詳情請見*How to create and use row factories*。

text_factory

A *callable* that accepts a `bytes` parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to `str`.

更多詳情請見*如何處理非 UTF-8 的文字編碼*。

total_changes

Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor 物件

A `Cursor` object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using `Connection.cursor()`, or by using any of the *connection shortcut methods*.

Cursor objects are *iterators*, meaning that if you `execute()` a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

`Cursor` 實例具有以下屬性和方法。

execute(*sql*, *parameters=()*, */*)

Execute a single SQL statement, optionally binding Python values using *placeholders*.

參數

- **sql** (`str`) -- 單一個 SQL 陳述式。
- **parameters** (*dict* | *sequence*) -- Python values to bind to placeholders in *sql*. A `dict` if named placeholders are used. A sequence if unnamed placeholders are used. See *How to use placeholders to bind values in SQL queries*.

引發

ProgrammingError -- When *sql* contains more than one SQL statement. When *named placeholders* are used and *parameters* is a sequence instead of a `dict`.

If `autocommit` is `LEGACY_TRANSACTION_CONTROL`, `isolation_level` is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

在 3.14 版的變更: `ProgrammingError` is emitted if *named placeholders* are used and *parameters* is a sequence instead of a `dict`.

Use `executescript()` to execute multiple SQL statements.

executemany(*sql*, *parameters*, */*)

For every item in *parameters*, repeatedly execute the *parameterized* DML (Data Manipulation Language) SQL statement *sql*.

Uses the same implicit transaction handling as `execute()`.

參數

- **sql** (`str`) -- A single SQL DML statement.
- **parameters** (*iterable*) -- An iterable of parameters to bind with the placeholders in *sql*. See *How to use placeholders to bind values in SQL queries*.

引發

ProgrammingError -- When *sql* contains more than one SQL statement or is not a DML statement, When *named placeholders* are used and the items in *parameters* are sequences instead of `dict`s.

範例：

```
rows = [
    ("row1",),
    ("row2",),
]
# cur 是一個 sqlite3.Cursor 物件
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

備🄵

Any resulting rows are discarded, including DML statements with RETURNING clauses.

在 3.14 版的變更: *ProgrammingError* is emitted if *named placeholders* are used and the items in *parameters* are sequences instead of *dict*s.

executescript(*sql_script*, /)

Execute the SQL statements in *sql_script*. If the *autocommit* is *LEGACY_TRANSACTION_CONTROL* and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script 必須是一個*string*。

範例：

```
# cur 是一個 sqlite3.Cursor 物件
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If *row_factory* is `None`, return the next row query result set as a *tuple*. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

fetchmany(*size=cursor.arraysize*)

Return the next set of rows of a query result as a *list*. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, *arraysize* determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one *fetchmany()* call to the next.

fetchall()

Return all (remaining) rows of a query result as a *list*. Return an empty list if no rows are available. Note that the *arraysize* attribute can affect the performance of this operation.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward; a *ProgrammingError* exception will be raised if any operation is attempted with the cursor.

setinputsizes(*sizes*, /)

Required by the DB-API. Does nothing in `sqlite3`.

setoutputsize(*size*, *column=None*, */*)

Required by the DB-API. Does nothing in `sqlite3`.

arraysize

Read/write attribute that controls the number of rows returned by *fetchmany()*. The default value is 1 which means a single row would be fetched per call.

connection

Read-only attribute that provides the SQLite database *Connection* belonging to the cursor. A *Cursor* object created by calling *con.cursor()* will have a *connection* attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the *execute()* method. For other statements, after *executemany()* or *executescript()*, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

> **備🄵**
>
> Inserts into `WITHOUT ROWID` tables are not recorded.

在 3.6 版的變更: 新增 `REPLACE` 陳述式的支援。

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements; is `-1` for other statements, including CTE (Common Table Expression) queries. It is only updated by the *execute()* and *executemany()* methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for `rowcount` to be updated.

row_factory

Control how a row fetched from this `Cursor` is represented. If `None`, a row is represented as a *tuple*. Can be set to the included *sqlite3.Row*; or a *callable* that accepts two arguments, a *Cursor* object and the `tuple` of row values, and returns a custom object representing an SQLite row.

Defaults to what *Connection.row_factory* was set to when the `Cursor` was created. Assigning to this attribute does not affect *Connection.row_factory* of the parent connection.

更多詳情請見*How to create and use row factories*。

Row 物件

class sqlite3.**Row**

A `Row` instance serves as a highly optimized *row_factory* for *Connection* objects. It supports iteration, equality testing, *len()*, and *mapping* access by column name and index.

Two `Row` objects compare equal if they have identical column names and values.

更多詳情請見*How to create and use row factories*。

keys()

Return a *list* of column names as *strings*. Immediately after a query, it is the first member of each tuple in *Cursor.description*.

在 3.5 版的變更: 新增對切片的支援。

Blob 物件

class sqlite3.**Blob**

在 3.11 版被加入.

A *Blob* instance is a *file-like object* that can read and write data in an SQLite BLOB. Call *len(blob)* to get the size (number of bytes) of the blob. Use indices and *slices* for direct access to the blob data.

Use the *Blob* as a *context manager* to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An *Error* (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1*, */*)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF (End of File) will be returned. When *length* is not specified, or is negative, *read()* will read until the end of the blob.

write(*data*, */*)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise *ValueError*.

tell()

Return the current access position of the blob.

seek(*offset*, *origin=os.SEEK_SET*, */*)

Set the current access position of the blob to *offset*. The *origin* argument defaults to *os.SEEK_SET* (absolute blob positioning). Other values for *origin* are *os.SEEK_CUR* (seek relative to the current position) and *os.SEEK_END* (seek relative to the blob’ s end).

PrepareProtocol 物件

class sqlite3.**PrepareProtocol**

The PrepareProtocol type's single purpose is to act as a **PEP 246** style adaption protocol for objects that can *adapt themselves* to *native SQLite types*.

例外

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is not currently raised by the `sqlite3` module, but may be raised by applications using `sqlite3`, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of *`Exception`*.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single `except` statement. `Error` is a subclass of *`Exception`*.

If the exception originated from within the SQLite library, the following two attributes are added to the exception:

sqlite_errorcode

The numeric error code from the SQLite API

在 3.11 版被加入.

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API

在 3.11 版被加入.

exception sqlite3.**InterfaceError**

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the `sqlite3` module. `InterfaceError` is a subclass of *`Error`*.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of *`Error`*.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of *`DatabaseError`*.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database's operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of *`DatabaseError`*.

exception sqlite3.**IntegrityError**

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of *`DatabaseError`*.

exception sqlite3.**InternalError**

Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. `InternalError` is a subclass of *`DatabaseError`*.

exception sqlite3.**ProgrammingError**

Exception raised for `sqlite3` API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed *`Connection`*. `ProgrammingError` is a subclass of *`DatabaseError`*.

exception sqlite3.**NotSupportedError**

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to `True` in *`create_function()`*, if the underlying SQLite library does not support deterministic functions. `NotSupportedError` is a subclass of *`DatabaseError`*.

SQLite and Python types

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

This is how SQLite types are converted to Python types by default:

SQLite type	Python type
`NULL`	`None`
`INTEGER`	*`int`*
`REAL`	*`float`*
`TEXT`	depends on *`text_factory`*, *`str`* by default
`BLOB`	*`bytes`*

The type system of the `sqlite3` module is extensible in two ways: you can store additional Python types in an SQLite database via *object adapters*, and you can let the `sqlite3` module convert SQLite types to Python types via *converters*.

Default adapters and converters (deprecated)

備🄵

The default adapters and converters are deprecated as of Python 3.12. Instead, use the *Adapter and converter recipes* and tailor them to your needs.

The deprecated default adapters and converters consist of:

- An adapter for *`datetime.date`* objects to *`strings`* in ISO 8601 format.
- An adapter for *`datetime.datetime`* objects to strings in ISO 8601 format.
- A converter for *declared* "date" types to *`datetime.date`* objects.
- A converter for declared "timestamp" types to *`datetime.datetime`* objects. Fractional parts will be truncated to 6 digits (microsecond precision).

備🄵

The default "timestamp" converter ignores UTC offsets in the database and always returns a naive *`datetime.datetime`* object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with *`register_converter()`*.

在 3.12 版之後被🄵用.

命令列介面

The `sqlite3` module can be invoked as a script, using the interpreter's `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

`-h, --help`
: Print CLI help.

`-v, --version`
: Print underlying SQLite library version.

在 3.12 版被加入.

12.6.3 How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a *`tuple`* of values to the second argument of the cursor's *`execute()`* method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a *sequence* whose length must match the number of placeholders, or a *`ProgrammingError`* is raised. For the named style, *parameters* must be an instance of a *`dict`* (or a subclass), which must contain keys for all named parameters; any extra items are ignored. Here's an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

備🄵

PEP 249 numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the *Python types SQLite natively understands*.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type *`PrepareProtocol`*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using *`register_adapter()`*.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

> **備🄵**
>
> Converter functions are **always** passed a *bytes* object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of *connect()*. There are three options:

- Implicit: set *detect_types* to *PARSE_DECLTYPES*
- Explicit: set *detect_types* to *PARSE_COLNAMES*
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
```

(繼續下一頁)

(繼續上一頁)

```
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.replace(tzinfo=None).isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the *execute()*, *executemany()*, and *executescript()* methods of the *Connection* class, your code can be written more concisely because you don't have to create the (often superfluous) *Cursor* objects explicitly. Instead, the *Cursor* objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a SELECT statement and iterate over it directly using only a single call on the *Connection* object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)
```

(繼續下一頁)

(繼續上一頁)

```
# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A *Connection* object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If *autocommit* is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if *autocommit* is `True`, the context manager does nothing.

> **備🄵**
>
> The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using *contextlib.closing()*.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- 以唯讀模式開🄴資料庫:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
>>> con.close()
```

- Do not implicitly create a new database file if it does not already exist; will raise *OperationalError* if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a *`tuple`*. If a `tuple` does not suit your needs, you can use the *`sqlite3.Row`* class or a custom *`row_factory`*.

While `row_factory` exists as an attribute both on the *`Cursor`* and the *`Connection`*, it is recommended to set *`Connection.row_factory`*, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]          # Access by index.
'Earth'
>>> row["name"]     # Access by name.
'Earth'
>>> row["RADIUS"]  # Column names are case-insensitive.
6378
>>> con.close()
```

> **備🄵**
>
> The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom *`row_factory`* that returns each row as a *`dict`*, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple`:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a *named tuple*:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
>>> con.close()
```

With some adjustments, the above recipe can be adapted to use a *`dataclass`*, or any other custom class, instead of a *`namedtuple`*.

如何處理非 UTF-8 的文字編碼

By default, `sqlite3` uses *`str`* to adapt SQLite values with the `TEXT` data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom *`text_factory`* to handle such cases.

Because of SQLite's flexible typing, it is not uncommon to encounter table columns with the `TEXT` data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let's assume we have a database with ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a *`Connection`* instance `con` connected to this database, we can decode the Latin-2 encoded text using this *`text_factory`*:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in `TEXT` table columns, you can use the following technique, borrowed from the unicode-howto:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

備🄵

The `sqlite3` module API does not support strings containing surrogates.

也參考

unicode-howto

12.6.4 解釋

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. *Transaction control via the autocommit attribute* is recommended, while *Transaction control via the isolation_level attribute* retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the *`Connection.autocommit`* attribute, which should preferably be set using the *autocommit* parameter of *`connect()`*.

It is suggested to set *autocommit* to `False`, which implies **PEP 249**-compliant transaction control. This means:

- `sqlite3` ensures that a transaction is always open, so *`connect()`*, *`Connection.commit()`*, and *`Connection.rollback()`* will implicitly open a new transaction (immediately after closing the pending one, for the latter two). `sqlite3` uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using `commit()`.
- Transactions should be rolled back explicitly using `rollback()`.
- An implicit rollback is performed if the database is *`close()`*-ed with pending changes.

Set *autocommit* to `True` to enable SQLite's autocommit mode. In this mode, *`Connection.commit()`* and *`Connection.rollback()`* have no effect. Note that SQLite's autocommit mode is distinct from the **PEP 249**-compliant *`Connection.autocommit`* attribute; use *`Connection.in_transaction`* to query the low-level SQLite autocommit mode.

Set *autocommit* to *`LEGACY_TRANSACTION_CONTROL`* to leave transaction control behaviour to the *`Connection.isolation_level`* attribute. See *Transaction control via the isolation_level attribute* for more information.

Transaction control via the `isolation_level` attribute

> **備F**
>
> The recommended way of controlling transactions is via the *`autocommit`* attribute. See *Transaction control via the autocommit attribute*.

If *`Connection.autocommit`* is set to *`LEGACY_TRANSACTION_CONTROL`* (the default), transaction behaviour is controlled using the *`Connection.isolation_level`* attribute. Otherwise, `isolation_level` has no effect.

If the connection attribute *`isolation_level`* is not `None`, new transactions are implicitly opened before *`execute()`* and *`executemany()`* executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the *`commit()`* and *`rollback()`* methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour —that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes –via the *`isolation_level`* attribute.

If *`isolation_level`* is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the *`in_transaction`* attribute.

The *`executescript()`* method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of *`isolation_level`*.

在 3.6 版的變更: `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

在 3.12 版的變更: The recommended way of controlling transactions is now via the *`autocommit`* attribute.

CHAPTER 13

資料壓縮與保存

本章中描述的模組支援使用 zlib、gzip、bzip2、lzma 和 zstd 演算法進行資料壓縮，以及建立 ZIP 和 tar 格式的存檔。另請參🄴*shutil* 模組提供的*Archiving operations*。

13.1 compression 套件

在 3.14 版被加入.

The `compression` package contains the canonical compression modules containing interfaces to several different compression algorithms. Some of these modules have historically been available as separate modules; those will continue to be available under their original names for compatibility reasons, and will not be removed without a deprecation cycle. The use of modules in `compression` is encouraged where practical.

- `compression.bz2` -- Re-exports *bz2*
- `compression.gzip` -- Re-exports *gzip*
- `compression.lzma` -- Re-exports *lzma*
- `compression.zlib` -- Re-exports *zlib*
- *compression.zstd* -- Wrapper for the Zstandard compression library

13.2 compression.zstd --- 與 Zstandard 格式相容的壓縮

在 3.14 版被加入.

原始碼： Lib/compression/zstd/__init__.py

This module provides classes and functions for compressing and decompressing data using the Zstandard (or *zstd*) compression algorithm. The zstd manual describes Zstandard as "a fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios." Also included is a file interface that supports reading and writing the contents of `.zst` files created by the **zstd** utility, as well as raw zstd compressed streams.

`compression.zstd` 模組包含：

- The *open()* function and *ZstdFile* class for reading and writing compressed files.
- The *ZstdCompressor* and *ZstdDecompressor* classes for incremental (de)compression.

- The *compress()* and *decompress()* functions for one-shot (de)compression.
- The *train_dict()* and *finalize_dict()* functions and the *ZstdDict* class to train and manage Zstandard dictionaries.
- The *CompressionParameter*, *DecompressionParameter*, and *Strategy* classes for setting advanced (de)compression parameters.

13.2.1 例外

exception compression.zstd.**ZstdError**

This exception is raised when an error occurs during compression or decompression, or while initializing the (de)compressor state.

13.2.2 Reading and writing compressed files

compression.zstd.**open**(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*, *encoding=None*, *errors=None*, *newline=None*)

Open a Zstandard-compressed file in binary or text mode, returning a *file object*.

The *file* argument can be either a file name (given as a *str*, *bytes* or *path-like* object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The mode argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'ab'` for appending, or `'xb'` for exclusive creation. These can equivalently be given as `'r'`, `'w'`, `'a'`, and `'x'` respectively. You may also open in text mode with `'rt'`, `'wt'`, `'at'`, and `'xt'` respectively.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see *DecompressionParameter* for detailed information about supported parameters. The *zstd_dict* argument is a *ZstdDict* instance to be used during decompression. When reading, if the *level* argument is not None, a `TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced decompression parameters; see *CompressionParameter* for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be non-None. The *zstd_dict* argument is a *ZstdDict* instance to be used during compression.

In binary mode, this function is equivalent to the *ZstdFile* constructor: `ZstdFile(file, mode, ...)`. In this case, the *encoding*, *errors*, and *newline* parameters must not be provided.

In text mode, a *ZstdFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line endings.

class compression.zstd.**ZstdFile**(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*)

Open a Zstandard-compressed file in binary mode.

A *ZstdFile* can wrap an already-open *file object*, or operate directly on a named file. The *file* argument specifies either the file object to wrap, or the name of the file to open (as a *str*, *bytes* or *path-like* object). If wrapping an existing file object, the wrapped file will not be closed when the *ZstdFile* is closed.

The *mode* argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'xb'` for exclusive creation, or `'ab'` for appending. These can equivalently be given as `'r'`, `'w'`, `'x'` and `'a'` respectively.

If *file* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see *DecompressionParameter* for detailed information about supported parameters. The *zstd_dict* argument is a *ZstdDict* instance to be used during decompression. When reading, if the *level* argument is not None, a `TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced decompression parameters; see *CompressionParameter* for detailed information about supported parameters. The *level* argument is the

compression level to use when writing compressed data. Only one of *level* or *options* may be passed. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

`ZstdFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> **備🄵**
>
> While calling `peek()` does not change the file position of the `ZstdFile`, it may change the position of the underlying file object (for example, if the `ZstdFile` was constructed by passing a file object for *file*).

mode

`'rb'` for reading and `'wb'` for writing.

name

The name of the Zstandard file. Equivalent to the `name` attribute of the underlying *file object*.

13.2.3 Compressing and decompressing data in memory

`compression.zstd.`**compress**(*data*, *level=None*, *options=None*, *zstd_dict=None*)

Compress *data* (a *bytes-like object*), returning the compressed data as a `bytes` object.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict()` can be used to generate a Zstandard dictionary.

`compression.zstd.`**decompress**(*data*, *zstd_dict=None*, *options=None*)

Decompress *data* (a *bytes-like object*), returning the uncompressed data as a `bytes` object.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

If *data* is the concatenation of multiple distinct compressed frames, decompress all of these frames, and return the concatenation of the results.

class `compression.zstd.`**ZstdCompressor**(*level=None*, *options=None*, *zstd_dict=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see the module-level function `compress()`.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level`

to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the *CompressionParameter* documentation.

The *zstd_dict* argument is an optional instance of *ZstdDict* containing trained data to improve compression efficiency. The function *train_dict()* can be used to generate a Zstandard dictionary.

compress(*data*, *mode=ZstdCompressor.CONTINUE*)

Compress *data* (a *bytes-like object*), returning a *bytes* object with compressed data if possible, or otherwise an empty `bytes` object. Some of *data* may be buffered internally, for use in later calls to `compress()` and *flush()*. The returned data should be concatenated with the output of any previous calls to *compress()*.

The *mode* argument is a *ZstdCompressor* attribute, either *CONTINUE*, *FLUSH_BLOCK*, or *FLUSH_FRAME*.

When all data has been provided to the compressor, call the *flush()* method to finish the compression process. If *compress()* is called with *mode* set to *FLUSH_FRAME*, *flush()* should not be called, as it would write out a new empty frame.

flush(*mode=ZstdCompressor.FLUSH_FRAME*)

Finish the compression process, returning a *bytes* object containing any data stored in the compressor's internal buffers.

The *mode* argument is a *ZstdCompressor* attribute, either *FLUSH_BLOCK*, or *FLUSH_FRAME*.

set_pledged_input_size(*size*)

Specify the amount of uncompressed data *size* that will be provided for the next frame. *size* will be written into the frame header of the next frame unless *CompressionParameter.content_size_flag* is `False` or `0`. A size of `0` means that the frame is empty. If *size* is `None`, the frame header will omit the frame size. Frames that include the uncompressed data size require less memory to decompress, especially at higher compression levels.

If *last_mode* is not *FLUSH_FRAME*, a *ValueError* is raised as the compressor is not at the start of a frame. If the pledged size does not match the actual size of data provided to *compress()*, future calls to `compress()` or *flush()* may raise *ZstdError* and the last chunk of data may be lost.

After *flush()* or *compress()* are called with mode *FLUSH_FRAME*, the next frame will not include the frame size into the header unless `set_pledged_input_size()` is called again.

CONTINUE

Collect more data for compression, which may or may not generate output immediately. This mode optimizes the compression ratio by maximizing the amount of data per block and frame.

FLUSH_BLOCK

Complete and write a block to the data stream. The data returned so far can be immediately decompressed. Past data can still be referenced in future blocks generated by calls to *compress()*, improving compression.

FLUSH_FRAME

Complete and write out a frame. Future data provided to *compress()* will be written into a new frame and *cannot* reference past data.

last_mode

The last mode passed to either *compress()* or *flush()*. The value can be one of *CONTINUE*, *FLUSH_BLOCK*, or *FLUSH_FRAME*. The initial value is *FLUSH_FRAME*, signifying that the compressor is at the start of a new frame.

class compression.zstd.**ZstdDecompressor**(*zstd_dict=None*, *options=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see the module-level function *decompress()*.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the *DecompressionParameter* documentation.

The *zstd_dict* argument is an instance of *ZstdDict* containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

> **備🄵**
>
> This class does not transparently handle inputs containing multiple compressed frames, unlike the *decompress()* function and *ZstdFile* class. To decompress a multi-frame input, you should use *decompress()*, *ZstdFile* if working with a *file object*, or multiple `ZstdDecompressor` instances.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If *max_length* is non-negative, the method returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the *needs_input* attribute will be set to `True`.

Attempting to decompress data after the end of a frame will raise a *ZstdError*. Any data found after the end of the frame is ignored and saved in the *unused_data* attribute.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b''`.

needs_input

`False` if the *decompress()* method can provide more decompressed data before requiring new compressed input.

13.2.4 Zstandard 字典

compression.zstd.**train_dict**(*samples*, *dict_size*)

Train a Zstandard dictionary, returning a *ZstdDict* instance. Zstandard dictionaries enable more efficient compression of smaller sizes of data, which is traditionally difficult to compress due to less repetition. If you are compressing multiple similar groups of data (such as similar files), Zstandard dictionaries can improve compression ratios and speed significantly.

The *samples* argument (an iterable of *bytes* objects), is the population of samples used to train the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. The Zstandard documentation suggests an absolute maximum of no more than 100 KB, but the maximum can often be smaller depending on the data. Larger dictionaries generally slow down compression, but improve compression ratios. Smaller dictionaries lead to faster compression, but reduce the compression ratio.

compression.zstd.**finalize_dict**(*zstd_dict*, /, *samples*, *dict_size*, *level*)

An advanced function for converting a "raw content" Zstandard dictionary into a regular Zstandard dictionary. "Raw content" dictionaries are a sequence of bytes that do not need to follow the structure of a normal Zstandard dictionary.

The *zstd_dict* argument is a *ZstdDict* instance with the *dict_content* containing the raw dictionary contents.

The *samples* argument (an iterable of *bytes* objects), contains sample data for generating the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. See *train_dict()* for suggestions on the maximum dictionary size.

The *level* argument (an integer) is the compression level expected to be passed to the compressors using this dictionary. The dictionary information varies for each compression level, so tuning for the proper compression level can make compression more efficient.

class compression.zstd.**ZstdDict**(*dict_content*, /, *, *is_raw=False*)

A wrapper around Zstandard dictionaries. Dictionaries can be used to improve the compression of many small chunks of data. Use *train_dict()* if you need to train a new dictionary from sample data.

The *dict_content* argument (a *bytes-like object*), is the already trained dictionary information.

The *is_raw* argument, a boolean, is an advanced parameter controlling the meaning of *dict_content*. `True` means *dict_content* is a "raw content" dictionary, without any format restrictions. `False` means *dict_content* is an ordinary Zstandard dictionary, created from Zstandard functions, for example, *train_dict()* or the external **zstd** CLI.

When passing a `ZstdDict` to a function, the `as_digested_dict` and `as_undigested_dict` attributes can control how the dictionary is loaded by passing them as the `zstd_dict` argument, for example, `compress(data, zstd_dict=zd.as_digested_dict)`. Digesting a dictionary is a costly operation that occurs when loading a Zstandard dictionary. When making multiple calls to compression or decompression, passing a digested dictionary will reduce the overhead of loading the dictionary.

表格 1: Difference for compression

	Digested dictionary	Undigested dictionary
Advanced parameters of the compressor which may be overridden by the dictionary's parameters	`window_log`, `hash_log`, `chain_log`, `search_log`, `min_match`, `target_length`, `strategy`, `enable_long_distance_matchin` `ldm_hash_log`, `ldm_min_match`, `ldm_bucket_size_log`, `ldm_hash_rate_log`, and some non-public parameters.	None
`ZstdDict` internally caches the dictionary	Yes. It's faster when loading a digested dictionary again with the same compression level.	No. If you wish to load an undigested dictionary multiple times, consider reusing a compressor object.

If passing a `ZstdDict` without any attribute, an undigested dictionary is passed by default when compressing and a digested dictionary is generated if necessary and passed by default when decompressing.

dict_content

The content of the Zstandard dictionary, a `bytes` object. It's the same as the *dict_content* argument in the `__init__` method. It can be used with other programs, such as the `zstd` CLI program.

dict_id

Identifier of the Zstandard dictionary, a non-negative int value.

Non-zero means the dictionary is ordinary, created by Zstandard functions and following the Zstandard format.

`0` means a "raw content" dictionary, free of any format restriction, used for advanced users.

> **備🄵**
>
> The meaning of `0` for `ZstdDict.dict_id` is different from the `dictionary_id` attribute to the *`get_frame_info()`* function.

as_digested_dict

Load as a digested dictionary.

as_undigested_dict

Load as an undigested dictionary.

13.2.5 Advanced parameter control

class compression.zstd.**CompressionParameter**

An *`IntEnum`* containing the advanced compression parameter keys that can be used when compressing data.

The *`bounds()`* method can be used on any attribute to get the valid values for that parameter.

Parameters are optional; any omitted parameter will have it's value selected automatically.

Example getting the lower and upper bound of *`compression_level`*:

```
lower, upper = CompressionParameter.compression_level.bounds()
```

Example setting the *`window_log`* to the maximum size:

```
_lower, upper = CompressionParameter.window_log.bounds()
options = {CompressionParameter.window_log: upper}
compress(b'venezuelan beaver cheese', options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a compression parameter. This method should be called on the attribute you wish to retrieve the bounds of. For example, to get the valid values for *`compression_level`*, one may check the result of `CompressionParameter.compression_level.bounds()`.

Both the lower and upper bounds are inclusive.

compression_level

A high-level means of setting other compression parameters that affect the speed and ratio of compressing data.

Regular compression levels are greater than `0`. Values greater than `20` are considered "ultra" compression and require more memory than other levels. Negative values can be used to trade off faster compression for worse compression ratios.

Setting the level to zero uses *`COMPRESSION_LEVEL_DEFAULT`*.

window_log

Maximum allowed back-reference distance the compressor can use when compressing data, expressed as power of two, `1 << window_log` bytes. This parameter greatly influences the memory usage of compression. Higher values require more memory but gain better compression values.

A value of zero causes the value to be selected automatically.

hash_log

Size of the initial probe table, as a power of two. The resulting memory usage is `1 << (hash_log+2)` bytes. Larger tables improve compression ratio of strategies <= *dfast*, and improve compression speed of strategies > *dfast*.

A value of zero causes the value to be selected automatically.

chain_log

Size of the multi-probe search table, as a power of two. The resulting memory usage is `1 << (chain_log+2)` bytes. Larger tables result in better and slower compression. This parameter has no effect for the *fast* strategy. It's still useful when using *dfast* strategy, in which case it defines a secondary probe table.

A value of zero causes the value to be selected automatically.

search_log

Number of search attempts, as a power of two. More attempts result in better and slower compression. This parameter is useless for *fast* and *dfast* strategies.

A value of zero causes the value to be selected automatically.

min_match

Minimum size of searched matches. Larger values increase compression and decompression speed, but decrease ratio. Note that Zstandard can still find matches of smaller size, it just tweaks its search algorithm to look for this size and larger. For all strategies < *btopt*, the effective minimum is `4`; for all strategies > *fast*, the effective maximum is `6`.

A value of zero causes the value to be selected automatically.

target_length

The impact of this field depends on the selected *Strategy*.

For strategies *btopt*, *btultra* and *btultra2*, the value is the length of a match considered "good enough" to stop searching. Larger values make compression ratios better, but compresses slower.

For strategy *fast*, it is the distance between match sampling. Larger values make compression faster, but with a worse compression ratio.

A value of zero causes the value to be selected automatically.

strategy

The higher the value of selected strategy, the more complex the compression technique used by zstd, resulting in higher compression ratios but slower compression.

也參考

Strategy

enable_long_distance_matching

Long distance matching can be used to improve compression for large inputs by finding large matches at greater distances. It increases memory usage and window size.

`True` or `1` enable long distance matching while `False` or `0` disable it.

Enabling this parameter increases default *window_log* to 128 MiB except when expressly set to a different value. This setting is enabled by default if `window_log` >= 128 MiB and the compression strategy >= *btopt* (compression level 16+).

ldm_hash_log

Size of the table for long distance matching, as a power of two. Larger values increase memory usage and compression ratio, but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_min_match

Minimum match size for long distance matcher. Larger or too small values can often decrease the compression ratio.

A value of zero causes the value to be selected automatically.

ldm_bucket_size_log

Log size of each bucket in the long distance matcher hash table for collision resolution. Larger values improve collision resolution but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_hash_rate_log

Frequency of inserting/looking up entries into the long distance matcher hash table. Larger values improve compression speed. Deviating far from the default value will likely result in a compression ratio decrease.

A value of zero causes the value to be selected automatically.

content_size_flag

Write the size of the data to be compressed into the Zstandard frame header when known prior to compressing.

This flag only takes effect under the following scenarios:

- Calling *compress()* for one-shot compression
- Providing all of the data to be compressed in the frame in a single *ZstdCompressor.compress()* call, with the *ZstdCompressor.FLUSH_FRAME* mode.
- Calling *ZstdCompressor.set_pledged_input_size()* with the exact amount of data that will be provided to the compressor prior to any calls to *ZstdCompressor.compress()* for the current frame. `ZstdCompressor.set_pledged_input_size()` must be called for each new frame.

All other compression calls may not write the size information into the frame header.

`True` or `1` enable the content size flag while `False` or `0` disable it.

checksum_flag

A four-byte checksum using XXHash64 of the uncompressed content is written at the end of each frame. Zstandard's decompression code verifies the checksum. If there is a mismatch a *ZstdError* exception is raised.

`True` or `1` enable checksum generation while `False` or `0` disable it.

dict_id_flag

When compressing with a *ZstdDict*, the dictionary's ID is written into the frame header.

`True` or `1` enable storing the dictionary ID while `False` or `0` disable it.

nb_workers

Select how many threads will be spawned to compress in parallel. When `nb_workers` > 0, enables multi-threaded compression, a value of `1` means "one-thread multi-threaded mode". More workers improve speed, but also increase memory usage and slightly reduce compression ratio.

A value of zero disables multi-threading.

job_size

Size of a compression job, in bytes. This value is enforced only when *nb_workers* >= 1. Each compression job is completed in parallel, so this value can indirectly impact the number of active threads.

A value of zero causes the value to be selected automatically.

overlap_log

Sets how much data is reloaded from previous jobs (threads) for new jobs to be used by the look behind window during compression. This value is only used when *nb_workers* >= 1. Acceptable values vary from 0 to 9.

- 0 means dynamically set the overlap amount
- 1 means no overlap
- 9 means use a full window size from the previous job

Each increment halves/doubles the overlap size. "8" means an overlap of `window_size/2`, "7" means an overlap of `window_size/4`, etc.

class compression.zstd.**DecompressionParameter**

An *IntEnum* containing the advanced decompression parameter keys that can be used when decompressing data. Parameters are optional; any omitted parameter will have it's value selected automatically.

The *bounds()* method can be used on any attribute to get the valid values for that parameter.

Example setting the *window_log_max* to the maximum size:

```
data = compress(b'Some very long buffer of bytes...')

_lower, upper = DecompressionParameter.window_log_max.bounds()

options = {DecompressionParameter.window_log_max: upper}
decompress(data, options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a decompression parameter. This method should be called on the attribute you wish to retrieve the bounds of.

Both the lower and upper bounds are inclusive.

window_log_max

The base-two logarithm of the maximum size of the window used during decompression. This can be useful to limit the amount of memory used when decompressing data. A larger maximum window size leads to faster decompression.

A value of zero causes the value to be selected automatically.

class compression.zstd.**Strategy**

An *IntEnum* containing strategies for compression. Higher-numbered strategies correspond to more complex and slower compression.

> **備🄵**
>
> The values of attributes of `Strategy` are not necessarily stable across zstd versions. Only the ordering of the attributes may be relied upon. The attributes are listed below in order.

The following strategies are available:

fast

dfast

greedy

lazy

lazy2

btlazy2

btopt

btultra

btultra2

13.2.6 Miscellaneous

compression.zstd.**get_frame_info**(*frame_buffer*)

Retrieve a *FrameInfo* object containing metadata about a Zstandard frame. Frames contain metadata related to the compressed data they hold.

class compression.zstd.**FrameInfo**

Metadata related to a Zstandard frame.

decompressed_size

The size of the decompressed contents of the frame.

dictionary_id

An integer representing the Zstandard dictionary ID needed for decompressing the frame. `0` means the dictionary ID was not recorded in the frame header. This may mean that a Zstandard dictionary is not needed, or that the ID of a required dictionary was not recorded.

compression.zstd.**COMPRESSION_LEVEL_DEFAULT**

The default compression level for Zstandard: `3`.

compression.zstd.**zstd_version_info**

Version number of the runtime zstd library as a tuple of integers (major, minor, release).

13.2.7 范例

讀取壓縮檔案：

```
from compression import zstd

with zstd.open("file.zst") as f:
    file_content = f.read()
```

Creating a compressed file:

```
from compression import zstd

data = b"Insert Data Here"
with zstd.open("file.zst", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
from compression import zstd

data_in = b"Insert Data Here"
data_out = zstd.compress(data_in)
```

Incremental compression:

```
from compression import zstd

comp = zstd.ZstdCompressor()
out1 = comp.compress(b"Some data\n")
```

(繼續下一頁)

(繼續上一頁)

```
out2 = comp.compress(b"Another piece of data\n")
out3 = comp.compress(b"Even more data\n")
out4 = comp.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
from compression import zstd

with open("myfile", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with zstd.open(f, "w") as zstf:
        zstf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using compression parameters:

```
from compression import zstd

options = {
   zstd.CompressionParameter.checksum_flag: 1
}
with zstd.open("file.zst", "w", options=options) as f:
    f.write(b"Mind if I squeeze in?")
```

13.3 `zlib` --- 相容於 `gzip` 的壓縮

對於需要資料壓縮的應用程式，此模組提供了能🄵使用 zlib 函式庫進行壓縮和解壓縮的函式。zlib 函式庫有自己的主頁 https://www.zlib.net。已知 Python 模組與早於 1.1.3 的 zlib 函式庫版本之間🄵不相容；1.1.3 存在安全漏洞，因此我們建議使用 1.1.4 或更新的版本。

zlib 的函式有很多選項，且通常需要按特定順序使用。本文件🄵不打算解🄵所有選項排列組合的效果；相關官方資訊，請參🄵 http://www.zlib.net/manual.html 上的 zlib 手🄵。

若要讀寫 `.gz` 文件，請參🄵*gzip* 模組。

該模組中可用的例外和函式是：

exception `zlib.`**`error`**

當壓縮和解壓縮發生錯誤時引發的例外。

`zlib.`**`adler32`**(*data*[, *value*])

計算 *data* 的 Adler-32 核對和 (checksum)。（Adler-32 核對和幾乎與 CRC32 一樣可靠，但計算速度更快。）結果是一個 unsigned（無符號的）32-bit 整數。如果有提供 *value*，則將其用作核對和的起始值，否則使用預設值 1。傳入 *value* 允許了於多個輸入的串聯 (concatenation) 上計算核對和。該演算法的加密🄵度不🄵高，不該用於身份驗證 (authentication) 或數位簽章 (digital signature)。由於該演算法是🄵核對和演算法而設計的，它🄵不適合作🄵通用的雜🄵演算法。

在 3.0 版的變更: 結果總是🄵 unsigned。

`zlib.`**`compress`**(*data*, */*, *level=-1*, *wbits=MAX_WBITS*)

壓縮 *data* 中的位元組，回傳一個包含壓縮資料的位元組物件。*level* 是從 `0` 到 `9` 或 `-1` 的整數，控制了壓縮的級🄵；`1` (Z_BEST_SPEED) 最快但壓縮程度較小，`9` (Z_BEST_COMPRESSION) 最慢但壓縮最多。`0` (Z_NO_COMPRESSION) 代表不壓縮。預設值🄵 `-1` (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 表示預設的速度和壓縮間折衷方案（目前相當於級🄵 6）。

wbits 引數控制了壓縮資料時所使用的歷史紀🄵緩衝區 (history buffer) 大小（或「視窗大小」），以及輸出中是否包含標題和尾末 (trailer)。它可以🄵用多個值的範圍，預設🄵 `15` (MAX_WBITS)：

- +9 到 +15：視窗大小的以二F底的對數，因此範圍在 512 到 32768 之間。較大的值會F生更佳的壓縮，但會F用更多的記憶體。生成輸出將包含特定於 zlib 的標頭和尾末。
- −9 到 −15：使用 *wbits* 的F對值作F視窗大小的對數，同時生成F有標頭或尾末核對和的原始輸出串流。
- +25 到 +31 = 16 + （9 到 15）：使用數值的最低 4 位元作F視窗大小的對數，同時在輸出中包含基本的 gzip 標頭和尾末核對和。

如果發生任何錯誤，則引發 `error` 例外。

在 3.6 版的變更: *level* 現在可以用作關鍵字參數。

在 3.11 版的變更: *wbits* 參數現在可用於設定視窗位元和壓縮型F。

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

回傳一個壓縮物件，用於壓縮不能一次全部放入記憶體中的資料串流。

level 是壓縮級F -- 從 `0` 到 `9` 或 `-1` 的整數。`1` (Z_BEST_SPEED) 最快但壓縮程度較小，而 `9` (Z_BEST_COMPRESSION) 最慢但壓縮最多。`0` (Z_NO_COMPRESSION) 代表不壓縮。預設值F `-1` (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 表示預設的速度和壓縮間折衷方案（目前相當於級F 6）。

method 代表壓縮演算法。目前唯一支援的值是 `DEFLATED`。

wbits 參數控制歷史紀F緩衝區的大小（或「視窗大小」），以及將使用的標頭和尾末格式。它與前面F述的 *compress()* 具有相同的含義。

memLevel 引數控制用於F部壓縮狀態的記憶體大小。有效值範圍F `1` 到 `9`。較高的值會使用更多的記憶體，但速度更快F生更小的輸出。

strategy 被用於調整壓縮演算法。可用的值F `Z_DEFAULT_STRATEGY`、`Z_FILTERED`、`Z_HUFFMAN_ONLY`、`Z_RLE` (zlib 1.2.0.1) 和 `Z_FIXED` (zlib 1.2.2.2)。

zdict 是事先定義好的的壓縮字典。這是一個位元組序列（例如一個 `bytes` 物件），其中包含預期在要壓縮的資料中頻繁出現的子序列。那些預期會最常見的子序列應該出現在字典的尾末。

在 3.3 版的變更: 新增 *zdict* 參數與支援關鍵字引數。

zlib.**crc32**(*data*[, *value*])

計算 *data* 的 CRC（Cyclic Redundancy Check，循環冗余核對）核對和，結果會是一個 unsigned 32-bit 整數。如果 *value* 存在，則將其用作核對和的起始值，否則使用預設值 0。傳入 *value* 允許在多個輸入的串聯上計算核對和。該演算法的加密F度不F高，不該用於身份驗證或數位簽章。由於該演算法是F核對和演算法而設計的，它F不適合作F通用的雜F演算法。

在 3.0 版的變更: 結果總是F unsigned。

zlib.**decompress**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

解壓縮 *data* 中的位元組，回傳包含未壓縮資料的位元組物件。*wbits* 參數依賴於 *data* 的格式，下面將進一步討論。如果有給定 *bufsize*，它會被用作輸出緩衝區的初始大小。如果發生任何錯誤，則引發 `error` 例外。

wbits 參數控制歷史紀F緩衝區的大小（或「視窗大小」），以及期望的標頭和尾末格式。它類似於 *compressobj()* 的參數，但接受更多範圍的值：

- +8 到 +15：視窗大小的以二F底的對數。輸入必須包括一個 zlib 標頭和尾末。
- 0：根據 zlib 標頭檔自動F定視窗大小。僅有在 zlib 1.2.3.5 或更新的版本支援。
- −8 to −15：使用 *wbits* 的F對值作F視窗大小的對數，輸入必須是F有標頭或尾末的原始串流。
- +24 到 +31 = 16 + （8 到 15）：取值的最低 4 位元作F視窗大小的對數，輸入必須包含 gzip 標頭和尾末。
- +40 到 +47 = 32 + （8 到 15）：使用值的最低 4 位元作F視窗大小的對數，F自動接受 zlib 或 gzip 格式。

當解壓縮一個串流時，視窗大小不得小於最初用於壓縮串流的大小；使用太小的值可能會導致*error* 例外。預設的 *wbits* 值對應於最大的視窗大小，🄵且需要包含 zlib 標頭和尾末。

bufsize 是用於保存解壓縮資料的緩衝區的初始大小。如果需要更多空間，緩衝區大小將根據需求來增加，因此你不需要讓該值完全剛好；調整它只會節省幾次對 `malloc()` 的呼叫。

在 3.6 版的變更: *wbits* 和 *bufsize* 可以用作關鍵字引數。

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])

回傳一個解壓縮物件，用於解壓縮不能一次全部放入記憶體的資料串流。

wbits 引數控制歷史紀🄵緩衝區的大小（或「視窗大小」），以及期望的標頭和尾末格式。它與前面🄵述的 *decompress()* 具有相同的含義。

zdict 參數指定是先定義好的壓縮字典。如果有提供，這必須與生成要解壓縮資料的壓縮器所使用的字典相同。

> **備🄵**
>
> 如果 *zdict* 是一個可變物件 (mutable object)（例如一個*bytearray*），你不能在呼叫*decompressobj()* 和第一次呼叫解壓縮器的 `decompress()` 方法之間修改它的🄵容。

在 3.3 版的變更: 新增 *zdict* 參數。

壓縮物件支援以下方法：

`Compress.`**`compress`**(*data*)

壓縮 *data*，回傳一個位元組物件，其中至少包含 *data* 中部分資料的壓縮資料。此資料應串聯到任何先前呼叫*compress()* 方法所🄵生的輸出。一些輸入可能會保存在🄵部緩衝區中以供後續處理。

`Compress.`**`flush`**([*mode*])

處理所有待處理的輸入，🄵回傳包含剩余壓縮輸出的位元組物件。*mode* 可以從以下常數中選擇：`Z_NO_FLUSH`、`Z_PARTIAL_FLUSH`、`Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、`Z_BLOCK` (zlib 1.2.3.4) 或 `Z_FINISH`，預設🄵 `Z_FINISH`。除了 `Z_FINISH` 之外，所有常數都允許壓縮更多的資料位元組字串，而 `Z_FINISH` 會完成壓縮串流🄵同時防止壓縮更多資料。在 *mode* 設定🄵 `Z_FINISH` 的情🄵下呼叫*flush()* 後，無法再次呼叫*compress()* 方法；唯一可行的作法是🄴除物件。

`Compress.`**`copy`**()

回傳壓縮物件的副本，這可用於有效壓縮一組共用初始前綴的資料。

在 3.8 版的變更: 於壓縮物件新增對*copy.copy()* 和*copy.deepcopy()* 的支援。

解壓縮物件支援以下方法和屬性：

`Decompress.`**`unused_data`**

一個位元組物件，它包含壓縮資料結束之後的任何位元組。也就是🄴，在包含壓縮資料的最後一個位元組可用之前，它會一直保持🄵 `b""`。如果整個位元組字串 (bytestring) 有包含壓縮資料，這會是 `b""`，也就是一個空位元組物件。

`Decompress.`**`unconsumed_tail`**

一個位元組物件，包含前一次*decompress()* 的呼叫因超出了未壓縮資料緩衝區的限制而🄵消耗掉的任何資料。zlib 機制尚未看到此資料，因此你必須將其（和可能有和它串聯的其他資料）反饋給後續的*decompress()* 方法呼叫以獲得正確的輸出。

`Decompress.`**`eof`**

一個布林值，代表是否已到達壓縮資料串流的尾末。

這使其能🄵區分有正確建構的壓縮串流和不完整或被截斷的串流。

在 3.3 版被加入.

Decompress.**decompress**(*data*, *max_length=0*)

解壓縮 *data* 🄵回傳一個位元組物件，其包含與 *string* 中至少與部分資料相對應的未壓縮資料。此資料應串聯到任何先前呼叫*decompress()* 方法所🄵生的輸出。一些輸入資料可能會保存在🄵部緩衝區中以供後續處理。

如果可選參數 *max_length* 不是零，則回傳值長度將不超過 *max_length*。這代表著🄵不是所有的已壓縮輸入都可以被處理；未使用的資料將被儲存在屬性*unconsumed_tail* 中。如果要繼續解壓縮，則必須將此位元組字串傳遞給後續對*decompress()* 的呼叫。如果 *max_length* 🄵零，則整個輸入會被解壓縮，🄵且*unconsumed_tail* 🄵空。

在 3.6 版的變更: *max_length* 可以用作關鍵字引數。

Decompress.**flush**([*length*])

處理所有待處理的輸入，🄵回傳包含剩余未壓縮輸出的位元組物件。呼叫*flush()* 後，無法再次呼叫*decompress()* 方法；唯一可行的方法是🄵除該物件。

可選參數 *length* 設定了輸出緩衝區的初始大小。

Decompress.**copy**()

回傳解壓物件的副本，這可用於在資料串流中途保存解壓縮器的狀態，以便在未來某個時間點加速對串流的隨機搜尋 (random seek)。

在 3.8 版的變更: 於解壓縮物件新增對*copy.copy()* 和*copy.deepcopy()* 支援。

有關正在使用的 zlib 函式庫版本資訊可通過以下常數獲得：

zlib.**ZLIB_VERSION**

用於建置模組的 zlib 函式庫版本字串。這可能與實際在執行環境 (runtime) 使用的 zlib 函式庫不同，後者以*ZLIB_RUNTIME_VERSION* 提供。

zlib.**ZLIB_RUNTIME_VERSION**

直譯器實際載入的 zlib 函式庫版本字串。

在 3.3 版被加入.

zlib.**ZLIBNG_VERSION**

The version string of the zlib-ng library that was used for building the module if zlib-ng was used. When present, the *ZLIB_VERSION* and *ZLIB_RUNTIME_VERSION* constants reflect the version of the zlib API provided by zlib-ng.

If zlib-ng was not used to build the module, this constant will be absent.

在 3.14 版被加入.

也參考

***gzip* 模組**
讀寫 **gzip** 格式的檔案。

http://www.zlib.net
zlib 函式庫首頁。

http://www.zlib.net/manual.html
zlib 手🄵解釋了函式庫中許多函式的語義和用法。

In case gzip (de)compression is a bottleneck, the python-isal package speeds up (de)compression with a mostly compatible API.

13.4 gzip --- gzip 檔案的支援

原始碼： Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs **gzip** and **gunzip** would.

The data compression is provided by the *zlib* module.

The *gzip* module provides the *GzipFile* class, as well as the *open()*, *compress()* and *decompress()* convenience functions. The *GzipFile* class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary *file object*.

Note that additional file formats which can be decompressed by the **gzip** and **gunzip** programs, such as those produced by **compress** and **pack**, are not supported by this module.

此模組定義了以下項目：

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a gzip-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'` for binary mode, or `'rt'`, `'at'`, `'wt'`, or `'xt'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

For binary mode, this function is equivalent to the *GzipFile* constructor: `GzipFile(filename, mode, compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *GzipFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

在 3.3 版的變更: Added support for *filename* being a file object, support for text mode, and the *encoding*, *errors* and *newline* arguments.

在 3.4 版的變更: 新增 `'x'`、`'xb'` 和 `'xt'` 模式的支援。

在 3.6 版的變更: 接受類路徑物件。

exception gzip.**BadGzipFile**

An exception raised for invalid gzip files. It inherits from *OSError*. *EOFError* and *zlib.error* can also be raised for invalid gzip files.

在 3.8 版被加入.

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

Constructor for the *GzipFile* class, which simulates most of the methods of a *file object*, with the exception of the *truncate()* method. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, an *io.BytesIO* object, or any other object which simulates a file. It defaults to `None`, in which case *filename* is opened to provide a file object.

When *fileobj* is not `None`, the *filename* argument is only used to be included in the **gzip** file header, which may include the original filename of the uncompressed file. It defaults to the filename of *fileobj*, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'`, or `'xb'`, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is `'rb'`. In future Python releases the mode of *fileobj* will not be used. It is better to always specify *mode* for writing.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use *open()* (or wrap your *GzipFile* with an *io.TextIOWrapper*).

The *compresslevel* argument is an integer from `0` to `9` controlling the level of compression; `1` is fastest and produces the least compression, and `9` is slowest and produces the most compression. `0` is no compression. The default is `9`.

The optional *mtime* argument is the timestamp requested by gzip. The time is in Unix format, i.e., seconds since 00:00:00 UTC, January 1, 1970. If *mtime* is omitted or `None`, the current time is used. Use *mtime* = 0 to generate a compressed stream that does not depend on creation time.

See below for the *mtime* attribute that is set when decompressing.

Calling a *GzipFile* object's `close()` method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an *io.BytesIO* object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the *io.BytesIO* object's *getvalue()* method.

GzipFile supports the *io.BufferedIOBase* interface, including iteration and the `with` statement. Only the *truncate()* method isn't implemented.

GzipFile 也提供了以下的方法和屬性:

peek(*n*)

Read *n* uncompressed bytes without advancing the file position. The number of bytes returned may be more or less than requested.

> **備🄵**
>
> While calling *peek()* does not change the file position of the *GzipFile*, it may change the position of the underlying file object (e.g. if the *GzipFile* was constructed with the *fileobj* parameter).

在 3.2 版被加入.

mode

`'rb'` for reading and `'wb'` for writing.

在 3.13 版的變更: In previous versions it was an integer `1` or `2`.

mtime

When decompressing, this attribute is set to the last timestamp in the most recently read header. It is an integer, holding the number of seconds since the Unix epoch (00:00:00 UTC, January 1, 1970). The initial value before reading any headers is `None`.

name

The path to the gzip file on disk, as a *str* or *bytes*. Equivalent to the output of *os.fspath()* on the original input path, with no other normalization, resolution or expansion.

在 3.1 版的變更: Support for the `with` statement was added, along with the *mtime* constructor argument and *mtime* attribute.

在 3.2 版的變更: Support for zero-padded and unseekable files was added.

在 3.3 版的變更: *io.BufferedIOBase.read1()* 方法現在已有實作。

在 3.4 版的變更: 新增 `'x'` 和 `'xb'` 模式的支援。

在 3.5 版的變更: Added support for writing arbitrary *bytes-like objects*. The *read()* method now accepts an argument of `None`.

在 3.6 版的變更: 接受類路徑物件。

在 3.9 版之後被🄵用: Opening *GzipFile* for writing without specifying the *mode* argument is deprecated.

在 3.12 版的變更: Remove the `filename` attribute, use the *name* attribute instead.

gzip.**compress**(*data*, *compresslevel=9*, ***, *mtime=0*)

Compress the *data*, returning a *bytes* object containing the compressed data. *compresslevel* and *mtime* have the same meaning as in the *GzipFile* constructor above, but *mtime* defaults to 0 for reproducible output.

在 3.2 版被加入.

在 3.8 版的變更: Added the *mtime* parameter for reproducible output.

在 3.11 版的變更: Speed is improved by compressing all data at once instead of in a streamed fashion. Calls with *mtime* set to `0` are delegated to *zlib.compress()* for better speed. In this situation the output may contain a gzip header "OS" byte value other than 255 "unknown" as supplied by the underlying zlib implementation.

在 3.13 版的變更: The gzip header OS byte is guaranteed to be set to 255 when this function is used as was the case in 3.10 and earlier.

在 3.14 版的變更: The *mtime* parameter now defaults to 0 for reproducible output. For the previous behaviour of using the current time, pass `None` to *mtime*.

`gzip.`**`decompress`**(*data*)

Decompress the *data*, returning a *bytes* object containing the uncompressed data. This function is capable of decompressing multi-member gzip data (multiple gzip blocks concatenated together). When the data is certain to contain only one member the *zlib.decompress()* function with *wbits* set to 31 is faster.

在 3.2 版被加入.

在 3.11 版的變更: Speed is improved by decompressing members at once in memory instead of in a streamed fashion.

13.4.1 用法范例

如何讀取壓縮檔案的範例:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

如何建立一個壓縮的 GZIP 檔案的範例:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

如何壓縮一個已存在的檔案的範例:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

如何壓縮一個二進位字串的範例:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

也參考

***zlib* 模組**
The basic data compression module needed to support the **`gzip`** file format.

In case gzip (de)compression is a bottleneck, the python-isal package speeds up (de)compression with a mostly compatible API.

13.4.2 命令列介面

The *gzip* module provides a simple command line interface to compress or decompress files.

Once executed the *gzip* module keeps the input file(s).

在 3.8 版的變更: Add a new command line interface with a usage. By default, when you will execute the CLI, the default compression level is 6.

命令列選項

file

如果未指定 *file*，則從 `sys.stdin` 讀取。

--fast

Indicates the fastest compression method (less compression).

--best

Indicates the slowest compression method (best compression).

-d, --decompress

解壓縮指定的檔案。

-h, --help

顯示幫助訊息。

13.5 bz2 --- bzip2 壓縮的支援

原始碼： Lib/bz2.py

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The `bz2` module contains:

- The `open()` function and `BZ2File` class for reading and writing compressed files.
- The `BZ2Compressor` and `BZ2Decompressor` classes for incremental (de)compression.
- The `compress()` and `decompress()` functions for one-shot (de)compression.

13.5.1 (De)compression of files

bz2.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a bzip2-compressed file in binary or text mode, returning a *file object*.

As with the constructor for `BZ2File`, the *filename* argument can be an actual filename (a `str` or `bytes` object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` or `'ab'` for binary mode, or `'rt'`, `'wt'`, `'xt'`, or `'at'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 1 to 9, as for the `BZ2File` constructor.

For binary mode, this function is equivalent to the `BZ2File` constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a `BZ2File` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s).

在 3.3 版被加入.

在 3.4 版的變更: The `'x'` (exclusive creation) mode was added.

在 3.6 版的變更: Accepts a *path-like object*.

class bz2.**BZ2File**(*filename*, *mode='r'*, ***, *compresslevel=9*)

Open a bzip2-compressed file in binary mode.

If *filename* is a `str` or `bytes` object, open the named file directly. Otherwise, *filename* should be a *file object*, which will be used to read or write the compressed data.

The *mode* argument can be either `'r'` for reading (default), `'w'` for overwriting, `'x'` for exclusive creation, or `'a'` for appending. These can equivalently be given as `'rb'`, `'wb'`, `'xb'` and `'ab'` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

If *mode* is `'w'` or `'a'`, *compresslevel* can be an integer between `1` and `9` specifying the level of compression: `1` produces the least compression, and `9` (default) produces the most compression.

If *mode* is `'r'`, the input file may be the concatenation of multiple compressed streams.

`BZ2File` provides all of the members specified by the *`io.BufferedIOBase`*, except for *`detach()`* and *`truncate()`*. Iteration and the `with` statement are supported.

`BZ2File` 也提供了以下方法和屬性:

peek([*n*])

Return buffered data without advancing the file position. At least one byte of data will be returned (unless at EOF). The exact number of bytes returned is unspecified.

> **備🄵**
>
> While calling *`peek()`* does not change the file position of the *`BZ2File`*, it may change the position of the underlying file object (e.g. if the *`BZ2File`* was constructed by passing a file object for *filename*).

在 3.3 版被加入.

fileno()

Return the file descriptor for the underlying file.

在 3.3 版被加入.

readable()

Return whether the file was opened for reading.

在 3.3 版被加入.

seekable()

Return whether the file supports seeking.

在 3.3 版被加入.

writable()

Return whether the file was opened for writing.

在 3.3 版被加入.

read1(*size=-1*)

Read up to *size* uncompressed bytes, while trying to avoid making multiple reads from the underlying stream. Reads up to a buffer's worth of data if size is negative.

Returns `b''` if the file is at EOF.

在 3.3 版被加入.

readinto(*b*)

Read bytes into *b*.

Returns the number of bytes read (0 for EOF).

在 3.3 版被加入.

mode

`'rb'` for reading and `'wb'` for writing.

在 3.13 版被加入.

name

The bzip2 file name. Equivalent to the `name` attribute of the underlying *file object*.

在 3.13 版被加入.

在 3.1 版的變更: Support for the `with` statement was added.

在 3.3 版的變更: Support was added for *filename* being a *file object* instead of an actual filename.

The `'a'` (append) mode was added, along with support for reading multi-stream files.

在 3.4 版的變更: The `'x'` (exclusive creation) mode was added.

在 3.5 版的變更: The `read()` method now accepts an argument of `None`.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.9 版的變更: The *buffering* parameter has been removed. It was ignored and deprecated since Python 3.0. Pass an open file object to control how the file is opened.

The *compresslevel* parameter became keyword-only.

在 3.10 版的變更: This class is thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in `gzip` and `lzma` have always been.

13.5.2 Incremental (de)compression

class bz2.**BZ2Compressor**(*compresslevel=9*)

Create a new compressor object. This object may be used to compress data incrementally. For one-shot compression, use the `compress()` function instead.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

compress(*data*)

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty byte string otherwise.

When you have finished providing data to the compressor, call the `flush()` method to finish the compression process.

flush()

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

class bz2.**BZ2Decompressor**

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot compression, use the `decompress()` function instead.

> **備🄵**
>
> This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `BZ2File`. If you need to decompress a multi-stream input with `BZ2Decompressor`, you must use a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the *needs_input* attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

在 3.5 版的變更: 新增 *max_length* 參數。

eof

`True` if the end-of-stream marker has been reached.

在 3.3 版被加入.

unused_data

Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be `b''`.

needs_input

`False` if the *decompress()* method can provide more decompressed data before requiring new uncompressed input.

在 3.5 版被加入.

13.5.3 One-shot (de)compression

`bz2.`**compress**(*data*, *compresslevel=9*)

Compress *data*, a *bytes-like object*.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

For incremental compression, use a *BZ2Compressor* instead.

`bz2.`**decompress**(*data*)

Decompress *data*, a *bytes-like object*.

If *data* is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a *BZ2Decompressor* instead.

在 3.3 版的變更: Support for multi-stream inputs was added.

13.5.4 用法范例

Below are some examples of typical usage of the *bz2* module.

Using *compress()* and *decompress()* to demonstrate round-trip compression:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

Using *BZ2Compressor* for incremental compression:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

The example above uses a very ”nonrandom” stream of data (a stream of `b"z"` chunks). Random data tends to compress poorly, while ordered, repetitive data usually yields a high compression ratio.

Writing and reading a bzip2-compressed file in binary mode:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after round-trip
True
```

13.6 `lzma` --- 使用 LZMA 演算法進行壓縮

在 3.3 版被加入.

原始碼： Lib/lzma.py

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the `.xz` and legacy `.lzma` file formats used by the **xz** utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the *bz2* module. Note that *LZMAFile* and *bz2.BZ2File* are *not* thread-safe, so if you need to use a single *LZMAFile* instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.**LZMAError**

This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

13.6.1 Reading and writing compressed files

`lzma.`**`open`**(*filename*, *mode='rb'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

Open an LZMA-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be either an actual file name (given as a `str`, `bytes` or *path-like* object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The *mode* argument can be any of `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` or `"ab"` for binary mode, or `"rt"`, `"wt"`, `"xt"`, or `"at"` for text mode. The default is `"rb"`.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for `LZMADecompressor`. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for `LZMACompressor`.

For binary mode, this function is equivalent to the `LZMAFile` constructor: `LZMAFile(filename, mode, ...)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a `LZMAFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s).

在 3.4 版的變更: 新增 `"x"`、`"xb"` 和 `"xt"` 模式的支援。

在 3.6 版的變更: Accepts a *path-like object*.

class `lzma.`**`LZMAFile`**(*filename=None*, *mode='r'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

Open an LZMA-compressed file in binary mode.

An `LZMAFile` can wrap an already-open *file object*, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a `str`, `bytes` or *path-like* object). When wrapping an existing file object, the wrapped file will not be closed when the `LZMAFile` is closed.

The *mode* argument can be either `"r"` for reading (default), `"w"` for overwriting, `"x"` for exclusive creation, or `"a"` for appending. These can equivalently be given as `"rb"`, `"wb"`, `"xb"` and `"ab"` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `"w"` does not truncate the file, and is instead equivalent to `"a"`.

When opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for `LZMADecompressor`. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for `LZMACompressor`.

`LZMAFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> **備🄵**
>
> While calling `peek()` does not change the file position of the `LZMAFile`, it may change the position of the underlying file object (e.g. if the `LZMAFile` was constructed by passing a file object for *filename*).

mode

'rb' for reading and 'wb' for writing.

在 3.13 版被加入.

name

The lzma file name. Equivalent to the *name* attribute of the underlying *file object*.

在 3.13 版被加入.

在 3.4 版的變更: 新增 "x" 和 "xb" 模式的支援。

在 3.5 版的變更: The *read()* method now accepts an argument of None.

在 3.6 版的變更: Accepts a *path-like object*.

13.6.2 Compressing and decompressing data in memory

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see *compress()*.

The *format* argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ: The .xz container format.**
  This is the default format.
- **FORMAT_ALONE: The legacy .lzma container format.**
  This format is more limited than .xz -- it does not support integrity checks or multiple filters.
- **FORMAT_RAW: A raw data stream, not using any container format.**
  This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using FORMAT_AUTO (see *LZMADecompressor*).

The *check* argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- CHECK_NONE: No integrity check. This is the default (and the only acceptable value) for FORMAT_ALONE and FORMAT_RAW.
- CHECK_CRC32: 32-bit Cyclic Redundancy Check.
- CHECK_CRC64: 64-bit Cyclic Redundancy Check. This is the default for FORMAT_XZ.
- CHECK_SHA256: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an *LZMAError* is raised.

The compression settings can be specified either as a preset compression level (with the *preset* argument), or in detail as a custom filter chain (with the *filters* argument).

The *preset* argument (if provided) should be an integer between 0 and 9 (inclusive), optionally OR-ed with the constant PRESET_EXTREME. If neither *preset* nor *filters* are given, the default behavior is to use PRESET_DEFAULT (preset level 6). Higher presets produce smaller output, but make the compression process slower.

> **備🄵**
>
> In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset 9 for example, the overhead for an *LZMACompressor* object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The *filters* argument (if provided) should be a filter chain specifier. See *Specifying custom filter chains* for details.

compress(*data*)

Compress *data* (a `bytes` object), returning a `bytes` object containing compressed data for at least part of the input. Some of *data* may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.

flush()

Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see `decompress()`.

The *format* argument specifies the container format that should be used. The default is `FORMAT_AUTO`, which can decompress both `.xz` and `.lzma` files. Other possible values are `FORMAT_XZ`, `FORMAT_ALONE`, and `FORMAT_RAW`.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an `LZMAError` if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is `FORMAT_RAW`, but should not be used for other formats. See *Specifying custom filter chains* for more information about filter chains.

> **備🄵**
>
> This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `LZMAFile`. To decompress a multi-stream input with `LZMADecompressor`, you must create a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the `needs_input` attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an `EOFError`. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

在 3.5 版的變更: 新增 *max_length* 參數。

check

The ID of the integrity check used by the input stream. This may be `CHECK_UNKNOWN` until enough of the input has been decoded to determine what integrity check it uses.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b""`.

needs_input

`False` if the *decompress()* method can provide more decompressed data before requiring new uncompressed input.

在 3.5 版被加入.

lzma.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Compress *data* (a *bytes* object), returning the compressed data as a *bytes* object.

See *LZMACompressor* above for a description of the *format*, *check*, *preset* and *filters* arguments.

lzma.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Decompress *data* (a *bytes* object), returning the uncompressed data as a *bytes* object.

If *data* is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See *LZMADecompressor* above for a description of the *format*, *memlimit* and *filters* arguments.

13.6.3 Miscellaneous

lzma.**is_check_supported**(*check*)

Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` and `CHECK_CRC32` are always supported. `CHECK_CRC64` and `CHECK_SHA256` may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

13.6.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- Compression filters:
  - `FILTER_LZMA1` (for use with `FORMAT_ALONE`)
  - `FILTER_LZMA2` (for use with `FORMAT_XZ` and `FORMAT_RAW`)
- Delta filter:
  - `FILTER_DELTA`
- Branch-Call-Jump (BCJ) filters:
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.

- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: `MODE_FAST` or `MODE_NORMAL`.
- `nice_len`: What should be considered a "nice length" for a match. This should be 273 or less.
- `mf`: What match finder to use -- `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, or `MF_BT4`.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.6.5 范例

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
```

(繼續下一頁)

(繼續上一頁)

```
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.7 `zipfile` --- 處理 ZIP 封存檔案

原始碼： Lib/zipfile/

ZIP 檔案格式是一種常見的封存 (archive) 與壓縮標準。本模組提供了建立、讀取、寫入、附加與列出 ZIP 檔案的工具。任何對本模組的進階使用都將需要對 PKZIP Application Note 中定義的格式有所理解。

本模組目前不處理多磁碟 ZIP 檔案。它可以處理使用 ZIP64 擴充（即大於 4 GiB 的 ZIP 檔案）的 ZIP 檔案。它支援解密 ZIP 封存檔案中的加密檔案，但目前無法建立加密檔案。解密速度極慢，因🄵它是在原生 Python 中實作，而不是 C。

本模組定義了以下項目：

exception zipfile.**BadZipFile**

對於損壞的 ZIP 檔案所引發的錯誤。

在 3.2 版被加入.

exception zipfile.**BadZipfile**

BadZipFile 的🄵名，🄵了與舊版 Python 相容。

在 3.2 版之後被🄵用.

exception zipfile.**LargeZipFile**

當 ZIP 檔案需要 ZIP64 功能但該功能未被🄵用時所引發的錯誤。

class zipfile.**ZipFile**

用於讀取和寫入 ZIP 檔案的類🄵。有關建構函式的詳細資訊，請參🄵*ZipFile 物件* 章節。

class zipfile.**Path**

實作了*pathlib.Path* 所提供介面子集的類🄵，包括完整的*importlib.resources.abc.Traversable* 介面。

在 3.8 版被加入.

class zipfile.**PyZipFile**

用於建立包含 Python 函式庫的 ZIP 封存檔案的類🄵。

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

用於表示封存檔案中成員資訊的類🄵。此類🄵的實例由*ZipFile* 物件的*getinfo()* 和*infolist()* 方法回傳。大多數*zipfile* 模組的使用者不需要建立這些實例，而只需使用本模組所建立的。*filename* 應🄵封存成員的完整名稱，而 *date_time* 應🄵一個包含六個欄位的元組，用以描述檔案的最後修改時間；這些欄位在*ZipInfo 物件* 章節中有所描述。

在 3.13 版的變更: 新增了一個公開的 `compress_level` 屬性，以公開先前受保護的 `_compresslevel`。🄵了向後相容，舊的受保護名稱仍可作🄵一個特性繼續運作。

_for_archive(*archive*)

Resolve the date_time, compression attributes, and external attributes to suitable defaults as used by *ZipFile.writestr()*.

Returns self for chaining.

在 3.14 版被加入.

zipfile.**is_zipfile**(*filename*)

如果 *filename* 根據其魔術數字（magic number）是一個有效的 ZIP 檔案，則回傳 `True`，否則回傳 `False`。*filename* 也可以是一個檔案或類檔案物件。

在 3.1 版的變更: 支援檔案和類檔案物件。

zipfile.**ZIP_STORED**

用於未壓縮封存成員的數值常數。

zipfile.**ZIP_DEFLATED**

用於常用的 ZIP 壓縮方法的數值常數。這需要*zlib* 模組。

zipfile.**ZIP_BZIP2**

用於 BZIP2 壓縮方法的數值常數。這需要*bz2* 模組。

在 3.3 版被加入.

zipfile.**ZIP_LZMA**

用於 LZMA 壓縮方法的數值常數。這需要*lzma* 模組。

在 3.3 版被加入.

zipfile.**ZIP_ZSTANDARD**

用於 Zstandard 壓縮方法的數值常數。這需要*compression.zstd* 模組。

備🄵

In APPNOTE 6.3.7, the method ID `20` was assigned to Zstandard compression. This was changed in APPNOTE 6.3.8 to method ID `93` to avoid conflicts, with method ID `20` being deprecated. For compatibility, the `zipfile` module reads both method IDs but will only write data with method ID `93`.

在 3.14 版被加入.

備🄵

ZIP 檔案格式規範自 2001 年起已包含對 bzip2 壓縮的支援、自 2006 年起支援 LZMA 壓縮、自 2020 年起支援 Zstandard 壓縮。然而某些工具（包括舊版的 Python）不支援這些壓縮方法，可能會完全拒🄵處理該 ZIP 檔案，或無法解壓縮個🄵檔案。

也參考

PKZIP Application Note
由 Phil Katz（該格式與所用演算法的創造者）所撰寫的關於 ZIP 檔案格式的文件。

Info-ZIP 首頁
關於 Info-ZIP 專案的 ZIP 封存程式和開發函式庫的資訊。

13.7.1 ZipFile 物件

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*, *metadata_encoding=None*)

開🄵一個 ZIP 檔案，其中 *file* 可以是一個檔案的路徑（一個字串）、一個類檔案物件或一個*path-like object*。

mode 參數應🄵 `'r'` 來讀取一個現有檔案、`'w'` 來清空🄵寫入一個新檔案、`'a'` 來附加到一個現有檔案，或 `'x'` 來獨🄵性地建立🄵寫入一個新檔案。如果 *mode* 🄵 `'x'` 且 *file* 指向一個現有檔案，將會引發`FileExistsError`。如果 *mode* 🄵 `'a'` 且 *file* 指向一個現有的 ZIP 檔案，則會將額外的檔案加入其中。如果 *file* 🄵非指向一個 ZIP 檔案，則一個新的 ZIP 封存檔案會被附加到該檔案之後。這適用於將一個 ZIP 封存檔案附加到另一個檔案（例如 `python.exe`）。如果 *mode* 🄵 `'a'` 且檔案完全不存在，它會被建立。如果 *mode* 🄵 `'r'` 或 `'a'`，該檔案應🄵可定位的 (seekable)。

compression 是寫入封存檔案時要使用的 ZIP 壓縮方法，應🄵`ZIP_STORED`、`ZIP_DEFLATED`、`ZIP_BZIP2`、`ZIP_LZMA` 或`ZIP_ZSTANDARD`；無法識🄵的值將導致引發`NotImplementedError`。如果指定了`ZIP_DEFLATED`、`ZIP_BZIP2`、`ZIP_LZMA` 或`ZIP_ZSTANDARD`，但對應的模組（`zlib`、`bz2`、`lzma` 或`compression.zstd`）不可用，則會引發`RuntimeError`。預設🄵`ZIP_STORED`。

如果 *allowZip64* 🄵 `True`（預設值），當 zipfile 大於 4 GiB 時，zipfile 將會建立使用 ZIP64 擴充的 ZIP 檔案。如果🄵 `false`，當 ZIP 檔案需要 ZIP64 擴充時，`zipfile` 將會引發一個例外。

compresslevel 參數控制寫入檔案到封存時使用的壓縮層級。當使用`ZIP_STORED` 或`ZIP_LZMA` 時，它🄵有效果。當使用`ZIP_DEFLATED` 時，接受整數 `0` 到 `9`（更多資訊請參🄵`zlib`）。當使用`ZIP_BZIP2` 時，接受整數 `1` 到 `9`（更多資訊請參🄵`bz2`）。當使用`ZIP_ZSTANDARD` 時，通常接受整數 `-131072` 到 `22`（更多關於檢索有效值及其含義，請參🄵`CompressionParameter.compression_level`）。

strict_timestamps 引數，當設🄵 `False` 時，允許壓縮早於 1980-01-01 的檔案，代價是將時間戳記設🄵 1980-01-01。對於晚於 2107-12-31 的檔案，也會有類似的行🄵，時間戳記同樣會被設🄵極限值。

當 mode 🄵 `'r'` 時，*metadata_encoding* 可以設🄵一個編解碼器的名稱，該編解碼器將被用來解碼元資料，例如成員名稱和 ZIP 🄵解。

如果檔案以 `'w'`、`'x'` 或 `'a'` 模式建立，然後在🄵有向封存檔案中加入任何檔案的情🄵下關閉，則適用於空封存檔案的 ZIP 結構將被寫入檔案中。

ZipFile 也是一個情境管理器，因此支援 `with` 陳述式。在範例中，*myzip* 在 `with` 陳述式的程式碼區塊執行完畢後會被關閉 --- 即使發生例外也是如此：

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

備🄵

metadata_encoding 是 ZipFile 的一個實例範圍設定。目前無法以每個成員🄵基礎來設定此項。

此屬性是針對舊版實作的變通方法，這些實作會以當前區域編碼或頁碼（主要在 Windows 上）🄵生帶有名稱的封存檔案。根據.ZIP 標准，元資料的編碼可以透過封存檔案標頭中的一個旗標指定🄵 IBM 頁碼（預設）或 UTF-8。該旗標的優先級高於 *metadata_encoding*，後者是 Python 特有的擴充。

在 3.2 版的變更: 新增`ZipFile` 作🄵情境管理器使用的能力。

在 3.3 版的變更: 新增對於`bzip2` 和`lzma` 壓縮的支援。

在 3.4 版的變更: ZIP64 擴充預設🄵🄵用。

在 3.5 版的變更: 新增了對寫入不可定位的 (unseekable) 串流的支援。新增了對 `'x'` 模式的支援。

在 3.6 版的變更: 先前，對於無法識🄵的壓縮值，會引發一個單純的`RuntimeError`。

在 3.6.2 版的變更: *file* 參數接受一個*path-like object*。

在 3.7 版的變更: 新增 *compresslevel* 參數。

在 3.8 版的變更: *strict_timestamps* 僅限關鍵字參數。

在 3.11 版的變更: 新增了在讀取 zipfile 目🄵和檔案標頭中的元資料時，可指定成員名稱編碼的支援。

ZipFile.**close**()

關閉封存檔案。你必須在結束程式前呼叫*close()*，否則必要的記🄵將不會被寫入。

ZipFile.**getinfo**(*name*)

回傳一個帶有關於封存成員 *name* 資訊的*ZipInfo* 物件。對一個目前不包含在封存檔案中的名稱呼叫*getinfo()* 將會引發*KeyError*。

ZipFile.**infolist**()

回傳一個串列，其中🄵每個封存成員包含一個*ZipInfo* 物件。如果開🄵的是一個現有的封存檔案，這些物件的順序與它們在磁碟上實際 ZIP 檔案中的條目順序相同。

ZipFile.**namelist**()

依名稱回傳封存成員的串列。

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, *, *force_zip64=False*)

以二進位類檔案物件的形式存取封存檔案中的一個成員。*name* 可以是封存檔案🄵的檔案名稱，或是一個*ZipInfo* 物件。*mode* 參數如果包含，則必須是 `'r'`（預設值）或 `'w'`。*pwd* 是用於解密加密 ZIP 檔案的密碼，🄵一個*bytes* 物件。

open() 也是一個情境管理器，因此支援 `with` 陳述式：

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

在 *mode* 🄵 `'r'` 時，該類檔案物件（`ZipExtFile`）是唯讀的，🄵提供以下方法：*read()*、*readline()*、*readlines()*、*seek()*、*tell()*、*__iter__()*、*__next__()*。這些物件可以獨立於 ZipFile 運作。

在 `mode='w'` 時，會回傳一個可寫的檔案控點 (file handle)，它支援*write()* 方法。當一個可寫的檔案控點開🄵時，嘗試讀取或寫入 ZIP 檔案中的其他檔案將會引發*ValueError*。

在這兩種情🄵下，該類檔案物件也具有 `name` 屬性，它相當於封存檔案🄵的檔案名稱，以及 `mode` 屬性，其值🄵 `'rb'` 或 `'wb'`，取🄵於輸入的模式。

寫入檔案時，如果檔案大小預先未知但可能超過 2 GiB，請傳入 `force_zip64=True` 以確保標頭格式能🄵支援大檔案。如果檔案大小預先已知，請建構一個設定了*file_size* 的*ZipInfo* 物件，🄵將其用作 *name* 參數。

> **備🄵**
>
> *open()*、*read()* 和*extract()* 方法可以接受一個檔名或一個*ZipInfo* 物件。當你嘗試讀取一個包含重🄵名稱成員的 ZIP 檔案時，你會發現這很有用。

在 3.6 版的變更: 移除了對 `mode='U'` 的支援。請使用*io.TextIOWrapper* 以*universal newlines* 模式讀取壓縮的文字檔。

在 3.6 版的變更: *ZipFile.open()* 現在可以用於以 `mode='w'` 選項將檔案寫入封存檔案。

在 3.6 版的變更: 在一個已關閉的 ZipFile 上呼叫*open()* 將會引發*ValueError*。先前，會引發一個*RuntimeError*。

在 3.13 版的變更: 🄵可寫的類檔案物件新增了 `name` 和 `mode` 屬性。可讀的類檔案物件的 `mode` 屬性值從 `'r'` 變更🄵 `'rb'`。

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)

從封存檔案中解壓縮一個成員到目前工作目🄵；*member* 必須是其完整名稱或一個*ZipInfo* 物件。其檔案資訊會被盡可能精確地解壓縮。*path* 指定一個不同的目🄵來解壓縮。*member* 可以是一個檔名或一個*ZipInfo* 物件。*pwd* 是用於加密檔案的密碼，🄵一個*bytes* 物件。

回傳建立的正規化路徑（一個目🄵或新檔案）。

> **備🄵**
>
> 如果成員檔名是🄵對路徑，磁碟機/UNC 共享點和開頭的（反）斜🄵將被移除，例如：在 Unix 上 `///foo/bar` 變成 `foo/bar`，在 Windows 上 `C:\foo\bar` 變成 `foo\bar`。🄵且成員檔名中所有的 `".."` 元件將被移除，例如：`../../foo../../ba..r` 變成 `foo../ba..r`。在 Windows 上，非法字元（`:`、`<`、`>`、`|`、`"`、`?` 和 `*`）會被底🄵（`_`）取代。

在 3.6 版的變更: 在一個已關閉的 ZipFile 上呼叫*extract()* 將會引發*ValueError*。先前，會引發一個*RuntimeError*。

在 3.6.2 版的變更: *path* 參數接受一個*path-like object*。

`ZipFile.`**`extractall`**(*path=None*, *members=None*, *pwd=None*)

從封存檔案中解壓縮所有成員到目前工作目🄵。*path* 指定一個不同的目🄵來解壓縮。*members* 是可選的，且必須是*namelist()* 回傳的串列的子集。*pwd* 是用於加密檔案的密碼，🄵一個*bytes* 物件。

> **警告**
>
> 🄵不要在未經事先檢查的情🄵下，從不受信任的來源解壓縮封存檔案。檔案有可能被建立在 *path* 之外，例如，成員具有以 `"/"` 開頭的🄵對檔名或帶有兩個點 `".."` 的檔名。本模組會試圖防止這種情🄵。請參🄵*extract()* 的🄵解。

在 3.6 版的變更: 在一個已關閉的 ZipFile 上呼叫*extractall()* 將會引發*ValueError*。先前，會引發一個*RuntimeError*。

在 3.6.2 版的變更: *path* 參數接受一個*path-like object*。

`ZipFile.`**`printdir`**()

將封存檔案的🄵容目🄵印出到 `sys.stdout`。

`ZipFile.`**`setpassword`**(*pwd*)

設定 *pwd*（一個*bytes* 物件）🄵解壓縮加密檔案的預設密碼。

`ZipFile.`**`read`**(*name*, *pwd=None*)

回傳封存檔案中 *name* 檔案的位元組。*name* 是封存檔案中檔案的名稱，或是一個*ZipInfo* 物件。封存檔案必須以讀取或附加模式開🄵。*pwd* 是用於加密檔案的密碼，🄵一個*bytes* 物件，如果指定，將會覆寫用*setpassword()* 設定的預設密碼。在一個使用*ZIP_STORED*、*ZIP_DEFLATED*、*ZIP_BZIP2*、*ZIP_LZMA* 或*ZIP_ZSTANDARD* 以外壓縮方法的 ZipFile 上呼叫*read()* 將會引發*NotImplementedError*。如果對應的壓縮模組不可用，也會引發一個錯誤。

在 3.6 版的變更: 在一個已關閉的 ZipFile 上呼叫*read()* 將會引發*ValueError*。先前，會引發一個*RuntimeError*。

`ZipFile.`**`testzip`**()

讀取封存檔案中的所有檔案，🄵檢查它們的 CRC 和檔案標頭。回傳第一個損壞檔案的名稱，否則回傳 `None`。

在 3.6 版的變更: 在一個已關閉的 ZipFile 上呼叫*testzip()* 將會引發*ValueError*。先前，會引發一個*RuntimeError*。

`ZipFile.`**`write`**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)

將名🄵 *filename* 的檔案寫入封存檔案，🄵給予它封存名稱 *arcname**（預設情🄵下，這將與 **filename* 相同，但不含磁碟機代號且移除了開頭的路徑分隔符）。如果給定，*compress_type* 將覆寫🄵建構函式中 *compression* 參數🄵新條目指定的值。同樣地，如果給定 *compresslevel*，它將覆寫建構函式。封存檔案必須以 `'w'`、`'x'` 或 `'a'` 模式開🄵。

> **備🄵**
>
> ZIP 檔案標准在歷史上🄴未指定元資料編碼，但🄴了互通性 (interoperability) 🄴烈建議使用 CP437（原始的 IBM PC 編碼）。近期的版本允許（僅）使用 UTF-8。在本模組中，如果成員名稱包含任何非 ASCII 字元，將會自動使用 UTF-8 來寫入。無法以 ASCII 或 UTF-8 以外的任何編碼寫入成員名稱。

> **備🄵**
>
> 封存名稱應相對於封存檔案的根目🄵，也就是🄴，它們不應以路徑分隔符開頭。

> **備🄵**
>
> 如果 `arcname`（或在未給定 `arcname` 時的 `filename`）包含一個空位元組，封存檔案中該檔案的名稱將在空位元組處被截斷。

> **備🄵**
>
> 檔名中開頭的斜🄴可能導致在 Windows 系統上的某些 zip 程式中無法開🄴該封存檔案。

在 3.6 版的變更: 在以 `'r'` 模式建立的 ZipFile 或一個已關閉的 ZipFile 上呼叫*`write()`* 將會引發*`ValueError`*。先前，會引發一個*`RuntimeError`*。

`ZipFile.`**`writestr`**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

將一個檔案寫入封存檔案。🄴容🄵 *data*，它可以是一個*`str`* 或一個*`bytes`* 實例；如果它是一個*`str`*，它會先被編碼🄴 UTF-8。*zinfo_or_arcname* 是它在封存檔案中將被賦予的檔案名稱，或是一個*`ZipInfo`* 實例。如果它是一個實例，則至少必須給定檔名、日期和時間。如果它是一個名稱，則日期和時間被設🄴目前的日期和時間。封存檔案必須以 `'w'`、`'x'` 或 `'a'` 模式開🄴。

如果給定，*compress_type* 將覆寫🄴建構函式中 *compression* 參數🄴新條目指定的值，或覆寫 *zinfo_or_arcname*（如果它是一個*`ZipInfo`* 實例）中的值。同樣地，如果給定 *compresslevel*，它將覆寫建構函式。

> **備🄵**
>
> 當傳遞一個*`ZipInfo`* 實例作🄴 *zinfo_or_arcname* 參數時，所使用的壓縮方法將是給定的*`ZipInfo`* 實例的 *compress_type* 成員中指定的方法。預設情🄴下，*`ZipInfo`* 建構函式將此成員設🄴*`ZIP_STORED`*。

在 3.2 版的變更: *compress_type* 引數。

在 3.6 版的變更: 在以 `'r'` 模式建立的 ZipFile 或一個已關閉的 ZipFile 上呼叫*`writestr()`* 將會引發*`ValueError`*。先前，會引發一個*`RuntimeError`*。

`ZipFile.`**`mkdir`**(*zinfo_or_directory*, *mode=511*)

在封存檔案🄵建立一個目🄵。如果 *zinfo_or_directory* 是一個字串，則會在封存檔案🄵以 *mode* 引數中指定的模式建立一個目🄵。然而，如果 *zinfo_or_directory* 是一個*`ZipInfo`* 實例，則 *mode* 引數會被忽略。

封存檔案必須以 `'w'`、`'x'` 或 `'a'` 模式開🄵。

在 3.11 版被加入.

以下資料屬性也可用：

ZipFile.**filename**

ZIP 檔案的名稱。

ZipFile.**debug**

要使用的偵錯輸出層級。可以從 0（預設，無輸出）設定到 3（最多輸出）。偵錯資訊會被寫入到 sys.stdout。

ZipFile.**comment**

與 ZIP 檔案關聯的🄵解，🄵一個*bytes* 物件。如果🄵一個以 'w'、'x' 或 'a' 模式建立的*ZipFile* 實例指派🄵解，其長度不應超過 65535 位元組。超過此長度的🄵解將被截斷。

13.7.2 Path 物件

class zipfile.**Path**(*root*, *at=''*)

從一個 root zipfile（它可以是一個*ZipFile* 實例或一個適合傳遞給*ZipFile* 建構函式的 file）建構一個 Path 物件。

at 指定此 Path 在 zipfile 中的位置，例如'dir/file.txt'、'dir/' 或''。預設🄵空字串，表示根目🄵。

> **備🄵**
>
> *Path* 類🄵不會清理 ZIP 封存檔案🄵的檔名。與*ZipFile.extract()* 和*ZipFile.extractall()* 方法不同，呼叫者有責任驗證或清理檔名，以防止路徑遍歷漏洞（例如，包含".." 或🄵對路徑的檔名）。在處理不受信任的封存檔案時，請考慮使用*os.path.abspath()* 解析檔名，🄵使用*os.path.commonpath()* 與目標目🄵進行核對。

Path 物件公開了*pathlib.Path* 物件的以下特性：

Path 物件可以使用 / 運算子或 joinpath 來遍歷。

Path.**name**

最後的路徑元件。

Path.**open**(*mode='r'*, ***, *pwd*, ****)

在目前路徑上呼叫*ZipFile.open()*。允許透過支援的模式'r'、'w'、'rb'、'wb' 以讀取或寫入、文字或二進位模式開🄵。當以文字模式開🄵時，位置引數和關鍵字引數會被傳遞給*io.TextIOWrapper*，否則會被忽略。pwd 是傳遞給*ZipFile.open()* 的 pwd 參數。

在 3.9 版的變更: 新增了對 open 的文字和二進位模式的支援。預設模式現在是文字模式。

在 3.11.2 版的變更: encoding 參數可以作🄵位置引數提供，而不會導致*TypeError*，就像在 3.9 中一樣。需要與未修補的 3.10 和 3.11 版本相容的程式碼，必須將所有*io.TextIOWrapper* 引數，包括 encoding，作🄵關鍵字傳遞。

Path.**iterdir**()

列舉目前目🄵的子項目。

Path.**is_dir**()

如果目前情境參照到一個目🄵，則回傳 True。

Path.**is_file**()

如果目前情境參照到一個檔案，則回傳 True。

Path.**is_symlink**()

如果目前情境參照到一個符號連結，則回傳 True。

在 3.12 版被加入.

在 3.13 版的變更: 先前，is_symlink 會無條件地回傳 False。

Path.**exists**()
: 如果目前情境參照到 zip 檔案中的一個檔案或目🄵，則回傳 `True`。

Path.**suffix**
: 最後一個元件中以點分隔的最後一部分（如果有的話）。這通常被稱🄵副檔名。

  在 3.11 版被加入: 新增*Path.suffix* 特性。

Path.**stem**
: 最後的路徑元件，不含其後綴。

  在 3.11 版被加入: 新增*Path.stem* 特性。

Path.**suffixes**
: 路徑的後綴串列，通常稱🄵副檔名。

  在 3.11 版被加入: 新增*Path.suffixes* 特性。

Path.**read_text**(*, **)
: 將目前檔案讀取🄵 unicode 文字。位置引數和關鍵字引數會被傳遞給*io.TextIOWrapper*（但 `buffer` 會根據情境隱式處理）。

  在 3.11.2 版的變更: `encoding` 參數可以作🄵位置引數提供，而不會導致*TypeError*，就像在 3.9 中一樣。需要與未修補的 3.10 和 3.11 版本相容的程式碼，必須將所有*io.TextIOWrapper* 引數，包括 `encoding`，作🄵關鍵字傳遞。

Path.**read_bytes**()
: 將目前檔案讀取🄵位元組。

Path.**joinpath**(**other*)
: 回傳一個新的 Path 物件，🄵將每個 *other* 引數加入。以下是等效的：

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

  在 3.10 版的變更: 在 3.10 之前，`joinpath` 🄵有文件記載，且只接受一個參數。

zipp 專案🄵舊版 Python 提供了最新的 path 物件功能的向後移植版本。使用 `zipp.Path` 來取代 `zipfile.Path` 以提早使用變更。

13.7.3 PyZipFile 物件

PyZipFile 建構函式接受與*ZipFile* 建構函式相同的參數，以及一個額外的參數 *optimize*。

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)
: 在 3.2 版的變更: 新增 *optimize* 參數。

  在 3.4 版的變更: ZIP64 擴充預設🄵🄵用。

  實例除了*ZipFile* 物件的方法之外，還有一個額外的方法：

  writepy(*pathname*, *basename=''*, *filterfunc=None*)
  : 搜尋 `*.py` 檔案，🄵將對應的檔案加入封存檔案。

    如果傳給*PyZipFile* 的 *optimize* 參數未被給定或🄵 `-1`，則對應的檔案是一個 `*.pyc` 檔案，如有必要則進行編譯。

    如果傳給*PyZipFile* 的 *optimize* 參數🄵 `0`、`1` 或 `2`，則只有具有該最佳化層級（參🄵*compile()*）的檔案會被加入封存檔案，如有必要則進行編譯。

    如果 *pathname* 是一個檔案，檔名必須以 `.py` 結尾，且只有（對應的 `*.pyc`）檔案會被加到頂層（🄵有路徑資訊）。如果 *pathname* 是一個不以 `.py` 結尾的檔案，將會引發*RuntimeError*。如果它是一個目🄵，且該目🄵不是一個套件目🄵，則所有 `*.pyc` 檔案都會被加到頂層。如果該目🄵是一個套件目🄵，則所有 `*.pyc` 都會以檔案路徑的形式被加到套件名稱下，且如果任何子目🄵是套件目🄵，所有這些都會以排序後的順序遞🄵地加入。

basename 僅供🄴部使用。

filterfunc，如果給定，必須是一個接受單一字串引數的函式。每個路徑（包括每個單獨的完整檔案路徑）在被加入封存檔案之前都會被傳遞給它。如果 *filterfunc* 回傳一個假值，該路徑將不會被加入，且如果它是一個目🄴，其🄴容將被忽略。例如，如果我們的測試檔案要嘛在 `test` 目🄴中，要嘛以字串 `test_` 開頭，我們可以使用一個 *filterfunc* 來排除它們：

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

writepy() 方法會建立帶有如下檔名的封存檔案：

```
string.pyc                      # 頂層名稱
test/__init__.pyc               # 套件目🄴
test/testall.pyc                # 模組 test.testall
test/bogus/__init__.pyc         # 子套件目🄴
test/bogus/myfile.pyc           # 子模組 test.bogus.myfile
```

在 3.4 版的變更: 新增 *filterfunc* 參數。

在 3.6.2 版的變更: *pathname* 參數接受一個*path-like object*。

在 3.7 版的變更: 遞🄴會對目🄴條目進行排序。

13.7.4 ZipInfo 物件

ZipInfo 類🄴的實例由*ZipFile* 物件的*getinfo()* 和*infolist()* 方法回傳。每個物件儲存關於 ZIP 封存檔案單一成員的資訊。

有一個類🄵方法可以🄴檔案系統中的檔案建立一個*ZipInfo* 實例：

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*, *, *strict_timestamps=True*)

🄴檔案系統上的一個檔案建構一個*ZipInfo* 實例，🄴將其加入 zip 檔案做准備。

filename 應🄴檔案系統上一個檔案或目🄴的路徑。

如果指定了 *arcname*，它將被用作封存檔案🄴的名稱。如果未指定 *arcname*，名稱將與 *filename* 相同，但會移除任何磁碟機代號和開頭的路徑分隔符。

strict_timestamps 引數，當設🄵 `False` 時，允許壓縮早於 1980-01-01 的檔案，代價是將時間戳記設🄴 1980-01-01。對於晚於 2107-12-31 的檔案，也會有類似的行🄴，時間戳記同樣會被設🄴極限值。

在 3.6 版被加入.

在 3.6.2 版的變更: *filename* 參數接受一個*path-like object*。

在 3.8 版的變更: 新增了 *strict_timestamps* 僅限關鍵字參數。

實例具有以下方法和屬性：

ZipInfo.**is_dir**()

如果此封存成員是一個目🄴，則回傳 `True`。

這會使用條目的名稱：目🄴應總是以 `/` 結尾。

在 3.6 版被加入.

ZipInfo.**filename**

封存檔案中檔案的名稱。

ZipInfo.**date_time**

封存成員的最後修改時間和日期。這是一個包含六個值的元組，代表 ZIP 檔案中中央目🄵的「最後 [修改] 檔案時間」和「最後 [修改] 檔案日期」欄位。

該元組包含：

索引	值
0	年（>= 1980）
1	月（從 1 開始）
2	日（從 1 開始）
3	時（從 0 開始）
4	分（從 0 開始）
5	秒（從 0 開始）

備🄵

The ZIP format supports multiple timestamp fields in different locations (central directory, extra fields for NTFS/UNIX systems, etc.). This attribute specifically returns the timestamp from the central directory. The central directory timestamp format in ZIP files does not support timestamps before 1980. While some extra field formats (such as UNIX timestamps) can represent earlier dates, this attribute only returns the central directory timestamp.

The central directory timestamp is interpreted as representing local time, rather than UTC time, to match the behavior of other zip tools.

ZipInfo.**compress_type**

封存成員的壓縮型🄵。

ZipInfo.**comment**

個🄵封存成員的🄵解，🄵一個*bytes* 物件。

ZipInfo.**extra**

擴充欄位資料。PKZIP Application Note 包含一些關於此*bytes* 物件中所含資料🄵部結構的🄵解。

ZipInfo.**create_system**

建立 ZIP 封存檔案的系統。

ZipInfo.**create_version**

建立 ZIP 封存檔案的 PKZIP 版本。

ZipInfo.**extract_version**

解壓縮封存檔案所需的 PKZIP 版本。

ZipInfo.**reserved**

必須🄵零。

ZipInfo.**flag_bits**

ZIP 旗標位元。

ZipInfo.**volume**

檔案標頭的磁碟空間編號。

ZipInfo.**internal_attr**

🄵部屬性。

ZipInfo.**external_attr**

外部檔案屬性。

ZipInfo.**header_offset**
: 到檔案標頭的位元組偏移量。

ZipInfo.**CRC**
: 未壓縮檔案的 CRC-32。

ZipInfo.**compress_size**
: 壓縮資料的大小。

ZipInfo.**file_size**
: 未壓縮檔案的大小。

13.7.5 命令列介面

zipfile 模組提供了一個簡單的命令列介面，用以與 ZIP 封存檔案互動。

如果你想建立一個新的 ZIP 封存檔案，請在*-c* 選項後指定其名稱，然後列出應包含的檔名：

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

傳遞一個目🄵也是可以的：

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果你想將一個 ZIP 封存檔案解壓縮到指定的目🄵，請使用*-e* 選項：

```
$ python -m zipfile -e monty.zip target-dir/
```

要列出 ZIP 封存檔案中的檔案，請使用*-l* 選項：

```
$ python -m zipfile -l monty.zip
```

命令列選項

-l <zipfile>

--list <zipfile>
: 列出 zipfile 中的檔案。

-c <zipfile> <source1> ... <sourceN>

--create <zipfile> <source1> ... <sourceN>
: 從來源檔案建立 zipfile。

-e <zipfile> <output_dir>

--extract <zipfile> <output_dir>
: 將 zipfile 解壓縮到目標目🄵。

-t <zipfile>

--test <zipfile>
: 測試 zipfile 是否有效。

--metadata-encoding <encoding>
: 🄵*-l*、*-e* 和*-t* 指定成員名稱的編碼。

  在 3.11 版被加入.

13.7.6 解壓縮的陷阱

zipfile 模組中的解壓縮可能會由於下面列出的一些陷阱而失敗。

來自檔案本身

解壓縮可能由於不正確的密碼 / CRC 校驗和 / ZIP 格式或不支援的壓縮方法 / 解密而失敗。

檔案系統限制

超出不同檔案系統的限制可能導致解壓縮失敗。例如目🄵條目中允許的字元、檔名長度、路徑名長度、單一檔案的大小以及檔案數量等。

資源限制

記憶體或磁碟空間不足會導致解壓縮失敗。例如，解壓縮炸彈（又稱 ZIP bomb）適用於 zipfile 函式庫，可能導致磁碟空間耗盡。

中斷

在解壓縮過程中斷，例如按下 control-C 或終止解壓縮過程，可能導致封存檔案解壓縮不完整。

解壓縮的預設行🄵

不了解預設的解壓縮行🄵可能導致非預期的解壓縮結果。例如，當兩次解壓縮同一個封存檔案時，它會不經詢問就覆寫檔案。

13.8 `tarfile` --- 讀取與寫入 tar 封存檔案

原始碼： Lib/tarfile.py

The *tarfile* module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the *zipfile* module to read or write `.zip` files, or the higher-level functions in *shutil*.

Some facts and figures:

- reads and writes *gzip*, *bz2*, *compression.zstd*, and *lzma* compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

在 3.3 版的變更: 新增對*lzma* 壓縮的支援。

在 3.12 版的變更: Archives are extracted using a *filter*, which makes it possible to either limit surprising/dangerous features, or to acknowledge that they are expected and the archive is fully trusted.

在 3.14 版的變更: Set the default extraction filter to *data*, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the filter strategy was equivalent to *fully_trusted*.

在 3.14 版的變更: 新增對 Zstandard 壓縮的支援（使用*compression.zstd*）。

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

Return a *TarFile* object for the pathname *name*. For detailed information on *TarFile* objects and the keyword arguments that are allowed, see *TarFile 物件*.

mode 必須是 `'filemode[:compression]'` 形式的字串，預設🄵 `'r'`。以下是所有可能的模式組合：

模式	操作
`'r'` 或 `'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	開🄵以讀取 gzip 壓縮。
`'r:bz2'`	開🄵以讀取 bzip2 壓縮。
`'r:xz'`	開🄵以讀取 lzma 壓縮。
`'r:zst'`	開🄵以讀取 Zstandard 壓縮。
`'x'` 或 `'x:'`	Create a tarfile exclusively without compression. Raise a *FileExistsError* exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise a *FileExistsError* exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise a *FileExistsError* exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise a *FileExistsError* exception if it already exists.
`'x:zst'`	Create a tarfile with Zstandard compression. Raise a *FileExistsError* exception if it already exists.
`'a'` 或 `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` 或 `'w:'`	Open for uncompressed writing.
`'w:gz'`	開🄵以用於 gzip 壓縮寫入。
`'w:bz2'`	開🄵以用於 bzip2 壓縮寫入。
`'w:xz'`	開🄵以用於 lzma 壓縮寫入。
`'w:zst'`	開🄵以用於 Zstandard 壓縮寫入。

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'x:gz'`, `'w|gz'`, `'w:bz2'`, `'x:bz2'`, `'w|bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For modes `'w:xz'`, `'x:xz'` and `'w|xz'`, *tarfile.open()* accepts the keyword argument *preset* to specify the compression level of the file.

For modes `'w:zst'`, `'x:zst'` and `'w|zst'`, *tarfile.open()* accepts the keyword argument *level* to specify the compression level of the file. The keyword argument *options* may also be passed, providing advanced Zstandard compression parameters described by *CompressionParameter*. The keyword argument *zstd_dict* can be passed to provide a *ZstdDict*, a Zstandard dictionary used to improve compression of smaller amounts of data.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a *read()* or *write()* method (depending on the *mode*) that works with bytes. *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin.buffer`, a socket *file object* or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see 範例. The currently possible modes:

模式	操作
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'r|zst'`	Open a Zstandard compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.
`'w|zst'`	開🄵以用於 Zstandard 壓縮寫入。

在 3.5 版的變更: The `'x'` (exclusive creation) mode was added.

在 3.6 版的變更: The *name* parameter accepts a *path-like object*.

在 3.12 版的變更: The *compresslevel* keyword argument also works for streams.

在 3.14 版的變更: The *preset* keyword argument also works for streams.

class tarfile.**TarFile**

Class for reading and writing tar archives. Do not use this class directly: use *tarfile.open()* instead. See *TarFile 物件*.

tarfile.**is_tarfile**(*name*)

Return *True* if *name* is a tar archive file, that the *tarfile* module can read. *name* may be a *str*, file, or file-like object.

在 3.9 版的變更: 對檔案和類檔案物件的支援。

tarfile 模組定義了以下例外：

exception tarfile.**TarError**

所有*tarfile* 例外的基底類🄵。

exception tarfile.**ReadError**

Is raised when a tar archive is opened, that either cannot be handled by the *tarfile* module or is somehow invalid.

exception tarfile.**CompressionError**

Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.**StreamError**

Is raised for the limitations that are typical for stream-like *TarFile* objects.

exception tarfile.**ExtractError**

Is raised for *non-fatal* errors when using *TarFile.extract()*, but only if *TarFile.errorlevel*== 2.

exception tarfile.**HeaderError**

Is raised by *TarInfo.frombuf()* if the buffer it gets is invalid.

exception tarfile.**FilterError**

Base class for members *refused* by filters.

tarinfo

Information about the member that the filter refused to extract, as *TarInfo*.

exception tarfile.**AbsolutePathError**

Raised to refuse extracting a member with an absolute path.

exception tarfile.**OutsideDestinationError**

Raised to refuse extracting a member outside the destination directory.

exception tarfile.**SpecialFileError**

Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.**AbsoluteLinkError**

Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.**LinkOutsideDestinationError**

Raised to refuse extracting a symbolic link pointing outside the destination directory.

exception tarfile.**LinkFallbackError**

Raised to refuse emulating a link (hard or symbolic) by extracting another archive member, when that member would be rejected by the filter location. The exception that was raised to reject the replacement member is available as `BaseException.__context__`.

在 3.14 版被加入.

The following constants are available at the module level:

tarfile.**ENCODING**

The default character encoding: `'utf-8'` on Windows, the value returned by *sys.getfilesystemencoding()* otherwise.

tarfile.**REGTYPE**

tarfile.**AREGTYPE**

一個普通檔案*type*。

tarfile.**LNKTYPE**

A link (inside tarfile) *type*.

tarfile.**SYMTYPE**

A symbolic link *type*.

tarfile.**CHRTYPE**

A character special device *type*.

tarfile.**BLKTYPE**

A block special device *type*.

tarfile.**DIRTYPE**

A directory *type*.

tarfile.**FIFOTYPE**

A FIFO special device *type*.

tarfile.**CONTTYPE**

A contiguous file *type*.

tarfile.**GNUTYPE_LONGNAME**

A GNU tar longname *type*.

tarfile.**GNUTYPE_LONGLINK**

A GNU tar longlink *type*.

tarfile.**GNUTYPE_SPARSE**

A GNU tar sparse file *type*.

Each of the following constants defines a tar archive format that the *tarfile* module is able to create. See section 支援的 *tar* 格式 for details.

tarfile.**USTAR_FORMAT**
: POSIX.1-1988 (ustar) 格式。

tarfile.**GNU_FORMAT**
: GNU tar 格式。

tarfile.**PAX_FORMAT**
: POSIX.1-2001 (pax) 格式。

tarfile.**DEFAULT_FORMAT**
: The default format for creating archives. This is currently *PAX_FORMAT*.

  在 3.8 版的變更: The default format for new archives was changed to *PAX_FORMAT* from *GNU_FORMAT*.

也參考

***zipfile* 模組**
: *zipfile* 標准模組的文件。

Archiving operations
: Documentation of the higher-level archiving facilities provided by the standard *shutil* module.

GNU tar manual, Basic Tar Format
: Documentation for tar archive files, including GNU tar extensions.

13.8.1 TarFile 物件

The *TarFile* object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a *TarInfo* object, see *TarInfo 物件* for details.

A *TarFile* object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *範例* section for a use case.

在 3.2 版被加入: 新增對情境管理協定的支援。

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*, *stream=False*)

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

備🄵

fileobj is not closed, when *TarFile* is closed.

format controls the archive format for writing. It must be one of the constants *USTAR_FORMAT*, *GNU_FORMAT* or *PAX_FORMAT* that are defined at module level. When reading, format will be automatically detected, even if different formats are present in a single archive.

The *tarinfo* argument can be used to replace the default *TarInfo* class with a different one.

If *dereference* is `False`, add symbolic and hard links to the archive. If it is `True`, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is `False`, treat an empty block as the end of the archive. If it is `True`, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

errorlevel controls how extraction errors are handled, see *the corresponding attribute*.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section *Unicode issues* for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is `PAX_FORMAT`.

If *stream* is set to `True` then while reading the archive info about files in the archive are not cached, saving memory.

在 3.2 版的變更: 使用 `'surrogateescape'` 作🄵 *errors* 引數的預設值。

在 3.5 版的變更: The `'x'` (exclusive creation) mode was added.

在 3.6 版的變更: The *name* parameter accepts a *path-like object*.

在 3.13 版的變更: 新增 *stream* 參數。

classmethod `TarFile.`**open**(...)

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

`TarFile.`**getmember**(*name*)

Return a `TarInfo` object for member *name*. If *name* can not be found in the archive, `KeyError` is raised.

> **備🄵**
>
> If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

`TarFile.`**getmembers**()

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.

`TarFile.`**getnames**()

Return the members as a list of their names. It has the same order as the list returned by `getmembers()`.

`TarFile.`**list**(*verbose=True*, *, *members=None*)

Print a table of contents to `sys.stdout`. If *verbose* is `False`, only the names of the members are printed. If it is `True`, output similar to that of **`ls -l`** is produced. If optional *members* is given, it must be a subset of the list returned by `getmembers()`.

在 3.5 版的變更: 新增 *members* 參數。

`TarFile.`**next**()

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return `None` if there is no more available.

`TarFile.`**extractall**(*path='.'*, *members=None*, *, *numeric_owner=False*, *filter=None*)

Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around

two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

The *filter* argument specifies how `members` are modified or rejected before extraction. See *Extraction filters* for details. It is recommended to set this explicitly only if specific *tar* features are required, or as `filter='data'` to support Python versions with a less secure default (3.13 and lower).

警告

Never extract archives from untrusted sources without prior inspection.

Since Python 3.14, the default (`data`) will prevent the most dangerous security issues. However, it will not prevent *all* unintended or insecure behavior. Read the *Extraction filters* section for details.

在 3.5 版的變更: 新增 *numeric_owner* 參數。

在 3.6 版的變更: The *path* parameter accepts a *path-like object*.

在 3.12 版的變更: 新增 *filter* 參數。

在 3.14 版的變更: The *filter* parameter now defaults to `'data'`.

`TarFile.`**`extract`**(*member*, *path=''*, *set_attrs=True*, ***, *numeric_owner=False*, *filter=None*)

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a `TarInfo` object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

The *numeric_owner* and *filter* arguments are the same as for `extractall()`.

備🄵

The `extract()` method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.

警告

Never extract archives from untrusted sources without prior inspection. See the warning for `extractall()` for details.

在 3.2 版的變更: 增加 *set_attrs* 參數。

在 3.5 版的變更: 新增 *numeric_owner* 參數。

在 3.6 版的變更: The *path* parameter accepts a *path-like object*.

在 3.12 版的變更: 新增 *filter* 參數。

`TarFile.`**`extractfile`**(*member*)

Extract a member from the archive as a file object. *member* may be a filename or a `TarInfo` object. If *member* is a regular file or a link, an `io.BufferedReader` object is returned. For all other existing members, `None` is returned. If *member* does not appear in the archive, `KeyError` is raised.

在 3.3 版的變更: Return an `io.BufferedReader` object.

在 3.13 版的變更: The returned `io.BufferedReader` object has the `mode` attribute which is always equal to `'rb'`.

TarFile.**errorlevel**: *int*

If *errorlevel* is 0, errors are ignored when using *TarFile.extract()* and *TarFile.extractall()*. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If 1 (the default), all *fatal* errors are raised as *OSError* or *FilterError* exceptions. If 2, all *non-fatal* errors are raised as *TarError* exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise *FilterError* for *fatal* errors and *ExtractError* for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

TarFile.**extraction_filter**

在 3.12 版被加入.

The *extraction filter* used as a default for the *filter* argument of *extract()* and *extractall()*.

The attribute may be None or a callable. String names are not allowed for this attribute, unlike the *filter* argument to *extract()*.

If extraction_filter is None (the default), extraction methods will use the *data* filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the TarFile class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or *site configuration*. To set a global default this way, a filter function needs to be wrapped in *staticmethod()* to prevent injection of a self argument.

在 3.14 版的變更: The default filter is set to *data*, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the default was equivalent to *fully_trusted*.

TarFile.**add**(*name*, *arcname=None*, *recursive=True*, *, *filter=None*)

Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to *False*. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a *TarInfo* object argument and returns the changed *TarInfo* object. If it instead returns *None* the *TarInfo* object will be excluded from the archive. See 範例 for an example.

在 3.2 版的變更: 新增 *filter* 參數。

在 3.7 版的變更: Recursion adds entries in sorted order.

TarFile.**addfile**(*tarinfo*, *fileobj=None*)

Add the *TarInfo* object *tarinfo* to the archive. If *tarinfo* represents a non zero-size regular file, the *fileobj* argument should be a *binary file*, and tarinfo.size bytes are read from it and added to the archive. You can create *TarInfo* objects directly, or by using *gettarinfo()*.

在 3.13 版的變更: *fileobj* must be given for non-zero-sized regular files.

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)

Create a *TarInfo* object from the result of *os.stat()* or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s *name* attribute, or the *name* argument. The name should be a text string.

You can modify some of the *TarInfo*'s attributes before you add it using *addfile()*. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as *size* may need modifying. This is the case for objects such as *GzipFile*. The *name* may also be modified, in which case *arcname* could be a dummy string.

在 3.6 版的變更: The *name* parameter accepts a *path-like object*.

TarFile.**close**()

Close the *TarFile*. In write mode, two finishing zero blocks are appended to the archive.

TarFile.**pax_headers**: *dict*

A dictionary containing key-value pairs of pax global headers.

13.8.2 TarInfo 物件

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods *getmember()*, *getmembers()* and *gettarinfo()*.

Modifying the objects returned by *getmember()* or *getmembers()* will affect all subsequent operations on the archive. For cases where this is unwanted, you can use *copy.copy()* or call the *replace()* method to create a modified copy in one step.

Several attributes can be set to `None` to indicate that a piece of metadata is unused or unknown. Different *TarInfo* methods handle `None` differently:

- The *extract()* or *extractall()* methods will ignore the corresponding metadata, leaving it set to a default.
- *addfile()* will fail.
- *list()* will print a placeholder string.

class tarfile.**TarInfo**(*name=''*)

Create a *TarInfo* object.

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)

Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

classmethod TarInfo.**fromtarfile**(*tarfile*)

Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

在 3.2 版的變更: 使用 `'surrogateescape'` 作🄵 *errors* 引數的預設值。

一個 `TarInfo` 物件具有以下公開資料屬性:

TarInfo.**name**: *str*

Name of the archive member.

TarInfo.**size**: *int*

Size in bytes.

TarInfo.**mtime**: *int* | *float*

Time of last modification in seconds since the *epoch*, as in *os.stat_result.st_mtime*.

在 3.12 版的變更: Can be set to `None` for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**mode**: *int*

Permission bits, as for *os.chmod()*.

在 3.12 版的變更: Can be set to `None` for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**type**

File type. *type* is usually one of these constants: *REGTYPE*, *AREGTYPE*, *LNKTYPE*, *SYMTYPE*, *DIRTYPE*, *FIFOTYPE*, *CONTTYPE*, *CHRTYPE*, *BLKTYPE*, *GNUTYPE_SPARSE*. To determine the type of a *TarInfo* object more conveniently, use the `is*()` methods below.

TarInfo.**linkname**: *str*

Name of the target file name, which is only present in *TarInfo* objects of type *LNKTYPE* and *SYMTYPE*.

For symbolic links (SYMTYPE), the *linkname* is relative to the directory that contains the link. For hard links (LNKTYPE), the *linkname* is relative to the root of the archive.

TarInfo.**uid**: *int*

User ID of the user who originally stored this member.

在 3.12 版的變更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gid**: *int*

Group ID of the user who originally stored this member.

在 3.12 版的變更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**uname**: *str*

User name.

在 3.12 版的變更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gname**: *str*

Group name.

在 3.12 版的變更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**chksum**: *int*

Header checksum.

TarInfo.**devmajor**: *int*

Device major number.

TarInfo.**devminor**: *int*

Device minor number.

TarInfo.**offset**: *int*

The tar header starts here.

TarInfo.**offset_data**: *int*

The file's data starts here.

TarInfo.**sparse**

Sparse member information.

TarInfo.**pax_headers**: *dict*

A dictionary containing key-value pairs of an associated pax extended header.

TarInfo.**replace**(*name=...*, *mtime=...*, *mode=...*, *linkname=...*, *uid=...*, *gid=...*, *uname=...*, *gname=...*, *deep=True*)

在 3.12 版被加入.

Return a *new* copy of the TarInfo object with the given attributes changed. For example, to return a TarInfo with the group name set to 'staff', use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. pax_headers and any custom attributes are shared with the original TarInfo object.

A *TarInfo* object also provides some convenient query methods:

TarInfo.**isfile**()
Return *True* if the *TarInfo* object is a regular file.

TarInfo.**isreg**()
Same as *isfile()*.

TarInfo.**isdir**()
如果它是一個目🄵，則回傳*True*。

TarInfo.**issym**()
如果它是一個符號連結，則回傳*True*。

TarInfo.**islnk**()
Return *True* if it is a hard link.

TarInfo.**ischr**()
如果它是一個字元裝置，則回傳*True*。

TarInfo.**isblk**()
如果它是一個區塊裝置，則回傳*True*。

TarInfo.**isfifo**()
如果它是一個 FIFO，則回傳*True*。

TarInfo.**isdev**()
如果它是一個字元裝置、區塊裝置或 FIFO，則回傳*True*。

13.8.3 Extraction filters

在 3.12 版被加入.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended -- and possibly malicious -- effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

警告

None of the available filters blocks *all* dangerous archive features. Never extract archives from untrusted sources without prior inspection. See also *Hints for further verification*.

也參考

PEP 706
Contains further motivation and rationale behind the design.

The *filter* argument to *TarFile.extract()* or *extractall()* can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See *tar_filter()* for details.
- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See *data_filter()* for details.

- `None` (default): Use *`TarFile.extraction_filter`*.

  If that is also `None` (the default), the `'data'` filter will be used.

  > 在 3.14 版的變更: The default filter is set to *`data`*. Previously, the default was equivalent to *`fully_trusted`*.

- A callable which will be called for each extracted member with a *TarInfo* describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

  ```
  filter(member: TarInfo, path: str, /) -> TarInfo | None
  ```

  The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

  - return a *`TarInfo`* object which will be used instead of the metadata in the archive, or
  - return `None`, in which case the member will be skipped, or
  - raise an exception to abort the operation or skip the member, depending on *`errorlevel`*. Note that when extraction is aborted, *`extractall()`* may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

`tarfile.`**`fully_trusted_filter`**(*member*, *path*)

Return *member* unchanged.

This implements the `'fully_trusted'` filter.

`tarfile.`**`tar_filter`**(*member*, *path*)

Implements the `'tar'` filter.

- Strip leading slashes (`/` and *`os.sep`*) from filenames.
- *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises *`AbsolutePathError`*.
- *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises *`OutsideDestinationError`*.
- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (*`S_IWGRP`* | *`S_IWOTH`*).

Return the modified `TarInfo` member.

`tarfile.`**`data_filter`**(*member*, *path*)

Implements the `'data'` filter. In addition to what `tar_filter` does:

- Normalize link targets (*`TarInfo.linkname`*) using *`os.path.normpath()`*. Note that this removes internal `..` components, which may change the meaning of the link if the path in `TarInfo.linkname` traverses symbolic links.
- *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

  This raises *`AbsoluteLinkError`* or *`LinkOutsideDestinationError`*.

  Note that such files are refused even on platforms that do not support symbolic links.
- *Refuse* to extract device files (including pipes). This raises *`SpecialFileError`*.
- For regular files, including hard links:
  - Set the owner read and write permissions (*`S_IRUSR`* | *`S_IWUSR`*).
  - Remove the group & other executable permission (*`S_IXGRP`* | *`S_IXOTH`*) if the owner doesn't have it (*`S_IXUSR`*).
- For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.

- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Note that this filter does not block *all* dangerous archive features. See *Hints for further verification* for details.

在 3.14 版的變更: Link targets are now normalized.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of *`FilterError`*. This will abort the extraction if *`TarFile.errorlevel`* is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a *`new temporary directory`* to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- Disallow symbolic links if you do not need the functionality.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, and so on).
- Check that filenames have expected extensions (discouraging files that execute when you "click on them", or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.
- *tarfile* does not protect against issues with "live" data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, but may be backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available:

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the `'data'` filter; *fail* if it is not available:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

  或：

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter; *warn* if it is not available:

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # 當不再需要時移除它
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.8.4 命令列介面

在 3.4 版被加入.

The `tarfile` module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the `-e` option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the *-l* option:

```
$ python -m tarfile -l monty.tar
```

命令列選項

-l <tarfile>

--list <tarfile>

List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>

--create <tarfile> <source1> ... <sourceN>

Create tarfile from source files.

-e <tarfile> [<output_dir>]

--extract <tarfile> [<output_dir>]

Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>

--test <tarfile>

Test whether the tarfile is valid or not.

-v, --verbose

Verbose output.

--filter <filtername>

Specifies the *filter* for `--extract`. See *Extraction filters* for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

13.8.5 范例

Reading examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

How to extract a subset of a tar archive with *TarFile.extractall()* using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

Writing examples

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to create and write an archive to stdout using *sys.stdout.buffer* in the *fileobj* parameter in *TarFile.add()*:

```
import sys
import tarfile
with tarfile.open("sample.tar.gz", "w|gz", fileobj=sys.stdout.buffer) as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to create an archive and reset the user information using the *filter* parameter in *TarFile.add()*:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.8.6 支援的 tar 格式

There are three tar formats that can be created with the *tarfile* module:

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.

- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally supported *ustar* format. It is the current default format for new archives.

  It extends the existing *ustar* format with extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.8.7 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in `tarfile` are controlled by the *encoding* and *errors* keyword arguments of the `TarFile` class.

encoding defines the character encoding to use for the metadata in the archive. The default value is `sys.getfilesystemencoding()` or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Error Handlers*. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see *File Names, Command Line Arguments, and Environment Variables*.

For `PAX_FORMAT` archives (the default), *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

CHAPTER 14

檔案格式

本章所描述的模組會剖析各種非標記類型語言且與電子郵件無關的雜項檔案格式。

14.1 csv --- CSV 檔案讀取及寫入

原始碼： Lib/csv.py

所謂的 CSV (Comma Separated Values) 檔案格式是試算表及資料庫中最常見的匯入、匯出檔案格式。在嘗試以 **RFC 4180** 中的標準化方式來描述格式之前，CSV 格式已經使用了許多年。由於缺少一個完善定義的標準，意味著各個不同的應用程式會在資料🄵生及銷🄵時有微妙的差🄵。這些不同之處使得從不同資料來源處理 CSV 檔案時會非常擾人。🄵管如此，雖然分隔符號和引號字元有所不同，整體的格式非常相似，可以寫個單一模組來高效率的操作這樣的資料，讓程式設計師可以隱藏讀取及寫入資料的細節。

csv 模組實作透過 class 去讀取、寫入 CSV 格式的表格資料。它讓程式設計師可以🄵出：「以 Excel 🄵首選🄵寫入該種格式的資料」或是「從 Excel 🄵生的檔案來讀取資料」，且無需知道這是 Excel 所使用的 CSV 格式等精確的細節。程式設計師也可以描述其他應用程式所理解的 CSV 格式或他們自行定義具有特殊意義的 CSV 格式。

csv 模組的 *reader* 及 *writer* 物件可以讀取及寫入序列。程式設計師也可以透過 *DictReader* 及 *DictWriter* class（類🄵）使用 dictionary （字典）讀取及寫入資料。

也參考

PEP 305 - CSV 檔案 API
Python Enhancement Proposal (PEP) 所提出的 Python 附加功能。

14.1.1 模組🄵容

csv 模組定義了以下函式：

`csv.`**`reader`**(*csvfile*, */*, *dialect='excel'*, ***fmtparams*)

回傳一個讀取器物件 *(reader object)* 🄵處理在指定的 *csvfile* 中的每一行，csvfile 必須是字串的可🄵代物件 (iterable of strings)，其中每個字串都要是讀取器所定義的 csv 格式，csvfile 通常是個類檔案物件或者 list。如果 *csvfile* 是個檔案物件，則需開🄵時使用 `newline=''`。[1] *dialect* 🄵一個可選填的參

[1] 如果 `newline=''` 🄵有被指定，則嵌入引號中的🄵行符號不會被正確直譯，使用 `\r\n` 行尾 (line ending) 的平台會寫入額外的 `\r`。自從 csv 模組有自己 (統一的) 🄵行處理方式，因此指定 `newline=''` 會永遠是安全的。

數，可以用𠀀特定的 CSV dialect（方言）定義一組參數。它可能𠀀*Dialect* 的一個子類𠀀 (subclass) 的實例或是由*list_dialects()* 函式回傳的多個字串中的其中之一。另一個可選填的關鍵字引數 *fmtparams* 可以在這個 dialect 中覆寫 (override) 個𠀀的格式化參數 (formatting parameter)。關於 dialect 及格式化參數的完整𠀀明，請見段落*Dialect 與格式參數*。

從 CSV 檔案讀取的每一列會回傳𠀀一個字串列表。除非格式選項*QUOTE_NONNUMERIC* 有被指定（在這個情𠀀之下，𠀀有引號的欄位都會被轉𠀀成浮點數），否則不會進行自動資料型𠀀轉𠀀。

一個簡短的用法範例：

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, */*, *dialect='excel'*, ***fmtparams*)

回傳一個寫入器物件 (writer object)，其負責在給定的類檔案物件 (file-like object) 上將使用者的資料轉𠀀𠀀分隔字串 (delimited string)。*csvfile* 可以𠀀具有*write()* method 的任何物件。若 *csvfile* 𠀀一個檔案物件，它應該使用 `newline=''` 開𠀀Page 619, 1 。*dialect* 𠀀一個可選填的參數，可以用𠀀特定的 CSV dialect 定義一組參數。它可能𠀀*Dialect* 的一個子類𠀀的實例或是由*list_dialects()* 函式回傳的多個字串中的其中之一。另一個可選填的關鍵字引數 *fmtparams* 可以在這個 dialect 中覆寫個𠀀的格式化參數。關於 dialect 及格式化參數的完整𠀀明，請見段落*Dialect 與格式參數*。𠀀了更容易與有實作 DB API 的模組互相接合，*None* 值會被寫成空字串。雖然這不是一個可逆的變𠀀，這使得 dump （傾印）SQL NULL 資料值到 CSV 檔案上就無需讓 `cursor.fetch*` 呼叫回傳的資料進行預處理 (preprocessing)。其余非字串的資料則會在寫入之前用*str()* 函式進行字串化 (stringify)。

一個簡短的用法範例：

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])

將 *dialect* 與 *name* 進行關聯 (associate)。*name* 必須𠀀字串。這個 dialect 可以透過傳遞*Dialect* 的子類𠀀進行指定；或是關鍵字引數 *fmtparams*；或是以上兩者皆是，𠀀透過關鍵字引數來覆寫 dialect 的參數。關於 dialect 及格式化參數的完整𠀀明，請見段落*Dialect 與格式參數*。

csv.**unregister_dialect**(*name*)

從 dialect 𠀀𠀀表 (registry) 中，𠀀除與 *name* 關聯的 dialect。若 *name* 如果不是𠀀𠀀的 dialect 名稱，則會𠀀生一個*Error*。

csv.**get_dialect**(*name*)

回傳一個與 *name* 關聯的 dialect。若 *name* 如果不是𠀀𠀀的 dialect 名稱，則會𠀀生一個*Error*。這個函式會回傳一個 immutable （不可變物件）*Dialect*。

csv.**list_dialects**()

回傳所有已𠀀𠀀的 dialect 名稱。

csv.**field_size_limit**([*new_limit*])

回傳目前的剖析器 (parser) 允許的最大字串大小。如果 *new_limit* 被給定，則會變成新的最大字串大小。

csv 模組定義了下列的類𠀀：

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

建立一個物件，其運作上就像一般的讀取器，但可以將每一列資訊 map （對映）到*dict* 中，可以透過選填的參數 *fieldnames* 設定 key。

參數 *fieldnames* 是一個*sequence*。如果 *fieldnames* 被省略了，檔案 *f* 中第一列的值會被當作欄位標題，且於結果中會被省略。如果 *fieldname* 有提供，它們就會被使用，且第一列會被包含在結果中。不管欄位標題是如何🄴定的，dictionary 都會保留原始的排序。

如果一列資料中的欄位比欄位標題還多，其余的資料及以 *restkey* （預設🄵 `None`）特指的欄位標題會放入列表當中🄴儲存。如果一個非空的 (non-blank) 列中的欄位比欄位標題還少，缺少的值則會填入 *restval* （預設🄵 `None`）的值。

所有其他選填的引數或關鍵字引數皆會傳遞至下層的*reader* 實例。

如果傳遞至 *fieldnames* 的引數是個🄴代器，則會被迫成🄵一個*list*。

在 3.6 版的變更: 回傳的列已成🄴型🄵 `OrderedDict`。

在 3.8 版的變更: 回傳的列已成🄴型🄵*dict*。

一個簡短的用法範例：

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)

建立一個物件，其運作上就像一般的寫入器，但可以將 dictionary map 到輸出的列上。參數 *fieldnames* 是一個鍵值的*sequence* 且可以辨識 dictionary 中傳遞至*writerow()* method 寫入至檔案 *f* 中的值。如果 dictionary 中缺少了 *fieldnames* 的鍵值，則會寫入選填的參數 *restval* 的值。如果傳遞至*writerow()* method 的 dictionary 包含了一個 *fieldnames* 中不存在的鍵值，選填的參數 *extrasaction* 可以指出該執行的動作。如果它被設定🄵 `'raise'`，預設會觸發*ValueError*。如果它被設定🄵 `'ignore'`，dictionary 中額外的值會被忽略。其他選填的引數或關鍵字引數皆會傳遞至下層的*writer* 實例。

請記得這不像類🄴*DictReader*，在類🄴*DictWriter* 中，參數 *fieldnames* 🄵不是選填的。

如果傳遞至 *fieldnames* 的引數是個🄴代器，則會被迫成🄵一個*list*。

一個簡短的用法範例：

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

類🄴*Dialect* 是一個容器類🄴，其屬性 (attribute) 包含如何處理雙引號、空白、分隔符號等資訊。由於缺少一個嚴謹的 CSV 技術規範，不同的應用程式會🄴出有巧妙不同的 CSV 資料。*Dialect* 實例定義了*reader* 以及*writer* 的實例該如何表示。

所有可用的*Dialect* 名稱會透過*list_dialects()* 回傳，且它們可以透過特定*reader* 及*writer* 類🄴的初始器 (initializer, `__init__`) 函式進行🄵🄵，就像這樣：

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.**excel**

類🄵*excel* 定義了透過 Excel 🄵生的 CSV 檔案的慣用屬性。它被🄵🄵的 dialect 名稱🄵 `'excel'`。

class csv.**excel_tab**

類🄵*excel_tab* 定義了透過 Excel 🄵生🄵以 Tab 作🄵分隔的 CSV 檔案的慣用屬性。它被🄵🄵的 dialect 名稱🄵 `'excel-tab'`。

class csv.**unix_dialect**

類🄵*unix_dialect* 定義了透過 UNIX 系統🄵生的 CSV 檔案的慣用屬性，🄵句話🄵，使用 `'\n'` 作🄵🄵行符號且所有欄位都被引號包覆起來。它被🄵🄵的 dialect 名稱🄵 `'unix'`。

在 3.2 版被加入.

class csv.**Sniffer**

類🄵*Sniffer* 被用來推斷 CSV 檔案的格式。

類🄵*Sniffer* 提供了兩個 method：

sniff(*sample*, *delimiters=None*)

分析給定的 *sample* 且回傳一個*Dialect* 子類🄵，反應出找到的格式參數。如果給定選填的參數 *delimiters*，它會被解釋🄵一個字串且含有可能、有效的分隔字元。

has_header(*sample*)

如果第一列的文字顯示將作🄵一系列的欄位標題，會分析 sample 文字（假定他是 CSV 格式）🄵回傳*True*。檢查每一欄時，會考慮是否滿足兩個關鍵標准其中之一，判斷 sample 是否包含標題：

- 第二列至第 n 列包含數字
- 第二列到第 n 列包含的字串中至少有一個值的長度與該行的假定標題的長度不同。

對第一列之後的二十個列進行取樣；如果超過一半的行及列滿足條件，則返回*True*。

> **備🄵**
>
> 此方法是一個粗略的🄵發，可能會🄵生🄵陽性及🄵陰性 (false positives and negatives)。

一個*Sniffer* 的使用範例：

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... 在這邊處理 CSV 檔案 ...
```

csv 模組定義了以下常數：

csv.**QUOTE_ALL**

引導*writer* 物件引用所有欄位。

csv.**QUOTE_MINIMAL**

引導*writer* 物件只引用包含特殊字元的欄位，例如：*delimiter*、*quotechar*、`'\r'`、`'\n'` 或是 *lineterminator* 的其他字元。

csv.**QUOTE_NONNUMERIC**

引導*writer* 物件引用所有非數字的欄位。

引導*reader* 物件轉🄵所有非引用的欄位🄵*float*。

> **備🄵**
>
> Some numeric types, such as *bool*, *Fraction*, or *IntEnum*, have a string representation that cannot be converted to *float*. They cannot be read in the *QUOTE_NONNUMERIC* and *QUOTE_STRINGS* modes.

csv.**QUOTE_NONE**

引導*writer* 物件不得引用欄位。當目前的 *delimiter*、*quotechar*、`'\r'`、`'\n'` 或是 *lineterminator* 的其他字元出現在輸出資料時，在他之前的字元是目前的 *escapechar*。如果🄵有設定 *escapechar*，若遇到任何字元需要逸出，寫入器則會引發*Error*。設定 *quotechar* 🄵 `None` 以防止逸出。

引導*reader* 物件不對引號進行特🄵處理。

csv.**QUOTE_NOTNULL**

引導*writer* 物件引用所有非 `None` 的欄位。這與*QUOTE_ALL* 相似，除非如果欄位值🄵 `None`，該欄位則被寫成空（🄵有引號）字串。

引導*reader* 物件將空（🄵有引號）欄位直譯 (interpret) 🄵 `None`，否則會和*QUOTE_ALL* 有相同的表現方式。

在 3.12 版被加入.

csv.**QUOTE_STRINGS**

引導*writer* 物件永遠在字串的欄位前後放置引號。這與*QUOTE_NONNUMERIC* 相似，除非如果欄位值🄵 `None`，該欄位則被寫成空（🄵有引號）字串。

引導*reader* 物件將空（🄵有引號）字串直譯🄵 `None`，否則會和*QUOTE_ALL* 有相同的表現方式。

在 3.12 版被加入.

csv 模組定義下列例外：

exception csv.**Error**

當偵測到錯誤時，任何函式都可以引發。

14.1.2 Dialect 與格式參數

🄵了讓指定輸入及輸出紀🄵的格式更方便，特定的格式化參數會被組成 dialect。一個 dialect 是*Dialect* class 的子類🄵，其包含多個描述 CSV 檔案格式的多個屬性。當建立*reader* 或*writer* 物件時，程式設計師可以指定一個字串或是一個*Dialect* 的子類🄵作🄵 dialect 參數。此外，或是作🄵替代，在 *dialect* 參數中，程式設計師可以指定個🄵的格式化參數，其與*Dialect* 類🄵定義的屬性具有相同的名字。

Dialect 支援下列屬性：

Dialect.**delimiter**

一個單一字元 (one-character) 的字串可已用來分割欄位。預設🄵 `','`。

Dialect.**doublequote**

控制 *quotechar* 的實例何時出現在欄位之中，🄵讓它們自己被放在引號之🄵。當屬性🄵*True*，字元會是雙引號。若🄵*False*，在 *quotechar* 之前會先使用 *escapechar* 作🄵前綴字。預設🄵*True*。

在輸出時，若 *doublequote* 是*False* 且逸出字元🄵有被設定，當一個引號在欄位中被發現時，*Error* 會被引發。

Dialect.**escapechar**

一個單一字元的字串，會被寫入器用來逸出需要逸出的字元：

- the *delimiter*, the *quotechar*, `'\r'`, `'\n'` and any of the characters in *lineterminator* are escaped if *quoting* is set to *QUOTE_NONE*;

- the *quotechar* is escaped if *doublequote* is `False`;
- *escapechar* 本身。

在讀取時，*escapechar* 會移除後面字元的任何特殊意義。預設🄵`None`，表示禁止逸出。

在 3.11 版的變更: *escapechar* 🄵空是不被接受的。

`Dialect.`**`lineterminator`**

由`writer`🄵生被用來分行的字串。預設🄵 `'\r\n'`。

> **備🄵**
>
> `reader` 是 hard-coded 辨🄵 `'\r'` or `'\n'` 作🄵行尾 (end-of-line)，🄵忽略 *lineterminator*。未來可能會改變這個行🄵。

`Dialect.`**`quotechar`**

一個單一字元的字串被用於引用包含特殊字元的欄位，像是 *delimiter*、*quotechar* 或是🄵行字元 (`'\r'`、`'\n'` 或是 *lineterminator* 的任一字元)。預設🄵 `'"'`。如果 *quoting* 設定🄵`QUOTE_NONE`，可以設定🄵 `None` 以防止逸出 `'"'`。

在 3.11 版的變更: *quotechar* 🄵空是不被允許的。

`Dialect.`**`quoting`**

控制 writer 何時🄵生引號，以及 reader 如何辨識引號。如 *quotechar* 不🄵 `None`，則可以使用任何*QUOTE_* 常數*且預設🄵`QUOTE_MINIMAL`。否則預設🄵`QUOTE_NONE`。

`Dialect.`**`skipinitialspace`**

若🄵`True`，在緊接著 *delimiter* 後的空格會被忽略。預設🄵`False`。當結合 `delimiter=' '` 與 `skipinitialspace=True` 時，不允許🄵有引號的空欄位。

`Dialect.`**`strict`**

若🄵 `True`，若有錯誤的 CSV 輸入則會引發`Error`。預設🄵 `False`。

14.1.3 讀取器物件

讀取器物件（`reader()` 函式回傳的`DictReader` 實例與物件）有下列公開方法 (public method)：

`csvreader.`**`__next__`**`()`

回傳一個列表🄵讀入器的可🄵代物件的下一列🄵容（若該物件是由`reader()` 回傳）或是一個 dict（若🄵`DictReader` 實例），會依據目前的`Dialect` 進行剖析。通常會用 `next(reader)` 來進行呼叫。

讀取器物件有下列公開屬性 (public attributes)：

`csvreader.`**`dialect`**

dialect 的唯讀🄵述，會被剖析器使用。

`csvreader.`**`line_num`**

來源🄵代器所讀取的行數。這與回傳的紀🄵數不同，因🄵可以進行跨行紀🄵。

DictReader 物件有下列公開屬性：

`DictReader.`**`fieldnames`**

若在建立物件時🄵有作🄵參數傳遞，這個屬性會在第一次存取之前或是第一筆資料被讀取之前進行初始化 (initialize)。

14.1.4 寫入器物件

`writer` 物件（`writer()` 函式回傳的`DictWriter` 實例與物件）有下列公開方法。對於`writer` 物件而言，一個列中必須🄵一個可🄵代的字串或是數字；對於`DictWriter` 物件而言，則必須🄵一個 dictionary，且可以對應欄位標題至字串或數字（會先透過`str()` 進行傳遞）。請注意，在寫入🄵數 (complex number) 時

會用小括號 (parens) 包起來。這可能在其他程式讀取 CSV 檔案時導致某些問題（假設他們完全支援🄵雜數字）。

csvwriter.**writerow**(*row*)

將參數 *row* 寫入至寫入器的檔案物件中，🄵依照目前的*Dialect* 進行格式化。回傳下層檔案物件 *write* 方法的回傳值。

在 3.5 版的變更: 新增對任意可🄵代物件 (arbitrary iterables) 的支援。

csvwriter.**writerows**(*rows*)

將 *rows* 中所有元素（🄵上述的一個可🄵代的 *row* 物件）寫入至寫入器的檔案物件中，🄵依照目前的 dialect 進行格式化。

寫入器物件有下列公開屬性：

csvwriter.**dialect**

dialect 的唯讀🄵述，會被寫入器使用。

DictWriter 物件有下列公開方法：

DictWriter.**writeheader**()

將具欄位標題的一列（於建構函式 (constructor) 中指定的）寫入至寫入器的檔案物件中，🄵依照目前的 dialect 進行格式化。回傳🄵部呼叫*csvwriter.writerow()* 的回傳值。

在 3.2 版被加入.

在 3.8 版的變更: *writeheader()* 現在也會回傳🄵部呼叫*csvwriter.writerow()* 的回傳值。

14.1.5 范例

最簡單的讀取 CSV 檔案範例：

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

讀取一個其他格式的檔案：

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

相對最簡單、可行的寫入範例🄵：

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

當*open()* 被使用於開🄵🄵讀取一個 CSV 檔案，該檔案會預設使用系統預設的編碼格式（請見*locale.getencoding()*），🄵解碼🄵 unicode。若要使用不同編碼格式進行檔案解碼，請使用 open 函式的 `encoding` 引數：

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

同理可以應用到使用不同編碼格式進行寫入：當開🄵輸出檔案時，指定 `encoding` 引數。

🄵🄵一個新的 dialect：

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

稍微進階的讀取器用法 -- 🄵取及回報錯誤：

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit(f'file {filename}, line {reader.line_num}: {e}')
```

而當模組無法直接支援剖析字串時，仍可以輕松的解🄵：

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

🄵解

14.2 `configparser` --- 設定檔剖析器

原始碼： Lib/configparser.py

This module provides the *`ConfigParser`* class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

備🄵

This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

也參考

***`tomllib`* 模組**
TOML is a well-specified format for application configuration files. It is specifically designed to be an improved version of INI.

***`shlex`* 模組**
Support for creating Unix shell-like mini-languages which can also be used for application configuration files.

***`json`* 模組**
The `json` module implements a subset of JavaScript syntax which is sometimes used for configuration, but does not support comments.

14.2.1 Quick Start

Let's take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described *in the following section*. Essentially, the file consists of sections, each of which contains keys with values. *configparser* classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, *outlined later*, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
```

(繼續下一頁)

(繼續上一頁)

```
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections[1]. Note also that keys in sections are case-insensitive and stored in lowercase[1].

It is possible to read several configurations into a single *`ConfigParser`*, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

This behaviour is equivalent to a *`ConfigParser.read()`* call with several files passed to the *filenames* parameter.

14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide *`getboolean()`*. This method is case-insensitive and recognizes Boolean values from `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` and `'1'`/`'0'`[1]. For example:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

[1] Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the *Customizing Parser Behaviour* section.

Apart from *getboolean()*, config parsers also provide equivalent *getint()* and *getfloat()* methods. You can register your own converters and customize the provided ones.[Page 628, 1]

14.2.3 Fallback Values

As with a dictionary, you can use a section's *get()* method to provide fallback values:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.example'`, we will always get the default, even if we specify a fallback:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level *get()* method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the *getint()*, *getfloat()* and *getboolean()* methods, for example:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 支援的 INI 檔案結構

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or : by default[Page 628, 1]). By default, section names are case sensitive but keys are not[Page 628, 1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it[Page 628, 1], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

By default, a valid section name can be any string that does not contain '\n'. To change this, see *ConfigParser.SECTCRE*.

The first section name may be omitted if the parser is configured to allow an unnamed top level section with `allow_unnamed_section=True`. In this case, the keys/values may be retrieved by *UNNAMED_SECTION* as in `config[UNNAMED_SECTION]`.

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default[Page 628, 1]). Comments may appear on their own on an otherwise empty line, possibly indented.[Page 628, 1]

舉例來🄵:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 Unnamed Sections

The name of the first section (or unique) may be omitted and values retrieved by the *UNNAMED_SECTION* attribute.

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed = configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

14.2.6 Interpolation of values

On top of the core functionality, *ConfigParser* supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.**BasicInterpolation**

The default implementation used by *ConfigParser*. It enables values to contain format strings which refer to other values in the same section, or values in the special default section[Page 628, 1]. Additional default values can be provided on initialization.

舉例來🄵:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character that needs to be escaped):
gain: 80%%
```

In the example above, *ConfigParser* with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With `interpolation` set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.**ExtendedInterpolation**

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be escaped):
cost: $$80
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
```

(繼續下一頁)

(繼續上一頁)

```
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.7 Mapping Protocol Access

在 3.2 版被加入.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of *configparser*, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

configparser objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the *MutableMapping* ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner[Page 628, 1]. E.g. `for option in parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a *KeyError*.
- `DEFAULTSECT` 不能從剖析器中移除：
  - 嘗試🄴除它會引發*ValueError*，
  - `parser.clear()` 會保持它不變，
  - `parser.popitem()` 永遠不會回傳它。
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.8 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. *configparser* goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- *defaults*, default value: `None`

  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

Hint: if you want to specify default values for a specific section, use *read_dict()* before you read the actual file.

- *dict_type*, default value: *dict*

  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the standard dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back.

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered. For example:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, default value: `False`

  Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by *configparser*. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, default value: `('=', ':')`

Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

See also the *space_around_delimiters* argument to *ConfigParser.write()*.

- *comment_prefixes*, default value: `('#', ';')`
- *inline_comment_prefixes*, default value: `None`

Comment prefixes are strings that indicate the start of a valid comment within a config file. *comment_prefixes* are used only on otherwise empty lines (optionally indented) whereas *inline_comment_prefixes* can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `';'` are used as prefixes for whole line comments.

在 3.2 版的變更: In previous versions of *configparser* behaviour matched `comment_prefixes=('#', ';')` and `inline_comment_prefixes=(';',)`.

Please note that config parsers don't support escaping of comment prefixes so using *inline_comment_prefixes* may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting *inline_comment_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, default value: `True`

When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using *read_file()*, *read_string()* or *read_dict()*). It is recommended to use strict parsers

in new applications.

在 3.2 版的變更: In previous versions of *configparser* behaviour matched `strict=False`.

- *empty_lines_in_values*, default value: `True`

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

  This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default_section*, default value: `configparser.DEFAULTSECT` (that is: `"DEFAULT"`)

  The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally called `"DEFAULT"` but this can be customized to point to any other valid section name. Some typical values include: `"general"` or `"common"`. The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: `configparser.BasicInterpolation`

  Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*. *RawConfigParser* has a default value of `None`.

- *converters*, default value: not set

  Config parsers provide option value getters that perform type conversion. By default *getint()*, *getfloat()*, and *getboolean()* are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{'decimal': decimal.Decimal}` would add `getdecimal()` on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0)`.

  If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

`ConfigParser.`**BOOLEAN_STATES**

By default when using *getboolean()*, config parsers consider the following values `True`: `'1'`, `'yes'`, `'true'`, `'on'` and the following values `False`: `'0'`, `'no'`, `'false'`, `'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
```

(繼續下一頁)

(繼續上一頁)

```
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `accept`/`reject` or `enabled`/`disabled`.

`ConfigParser.`**`optionxform`**(*option*)

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

備F

The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

`ConfigParser.`**`SECTCRE`**

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[ larch ]` will be read as a section of name `" larch "`. Override this attribute if that's unsuitable. For example:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
```

(繼續下一頁)

(繼續上一頁)

```
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

> **備🄵**
>
> While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.9 Legacy API Examples

Mainly because of backwards compatibility concerns, *configparser* provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use *ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

14.2.10 ConfigParser 物件

class configparser.**ConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be

indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising *DuplicateSectionError* or *DuplicateOptionError*. When *empty_lines_in_values* is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed at runtime using the `default_section` instance attribute. This won't re-evaluate an already parsed config file, but will be used when writing parsed settings to a new config file.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*.

All option names used in interpolation will be passed through the *optionxform()* method just like any other option name reference. For example, using the default implementation of *optionxform()* (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

When *allow_unnamed_section* is `True` (default: `False`), the first section name can be omitted. See the *"Unnamed Sections" section*.

It is possible to read several configurations into a single *ConfigParser*, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

在 3.1 版的變更: 預設的 *dict_type* 是*collections.OrderedDict*。

在 3.2 版的變更: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

在 3.5 版的變更: 新增 *converters* 引數。

在 3.7 版的變更: The *defaults* argument is read with *read_dict()*, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

在 3.8 版的變更: The default *dict_type* is *dict*, since it now preserves insertion order.

在 3.13 版的變更: Raise a *MultilineContinuationError* when *allow_no_value* is `True`, and a key without a value is continued with an indented line.

在 3.13 版的變更: 新增 *allow_unnamed_section* 引數。

defaults()

Return a dictionary containing the instance-wide defaults.

sections()

Return a list of the sections available; the *default section* is not included in the list.

add_section(*section*)

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. The name of the section must be a string; if not, `TypeError` is raised.

在 3.2 版的變更: Non-string section names raise `TypeError`.

has_section(*section*)

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)

Return a list of options available in the specified *section*.

has_option(*section*, *option*)

If the given *section* exists, and contains the given *option*, return `True`; otherwise return `False`. If the specified *section* is `None` or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)

Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a `bytes` object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `read_file()` before calling `read()` for any optional files:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

在 3.2 版的變更: Added the *encoding* parameter. Previously, all files were read using the default encoding for `open()`.

在 3.6.1 版的變更: The *filenames* parameter accepts a *path-like object*.

在 3.7 版的變更: The *filenames* parameter accepts a `bytes` object.

read_file(*f*, *source=None*)

Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is `'<???>'`.

在 3.2 版被加入: 取代 `readfp()`。

read_string(*string*, *source='<string>'*)

Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

在 3.2 版被加入.

read_dict(*dictionary*, *source='<dict>'*)

Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

在 3.2 版被加入.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

在 3.2 版的變更: Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to an integer. See *get()* for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a floating-point number. See *get()* for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise *ValueError*. See *get()* for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)

items(*section*, *raw=False*, *vars=None*)

When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the *get()* method.

在 3.8 版的變更: Items present in *vars* no longer appear in the result. The previous behaviour mixed actual parser options with variables provided for interpolation.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. *option* and *value* must be strings; if not, *TypeError* is raised.

write(*fileobject*, *space_around_delimiters=True*)

Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future *read()* call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

在 3.14 版的變更: Raises InvalidWriteError if this would write a representation which cannot be accurately parsed by a future *read()* call from this parser.

> **備🄵**
>
> Comments in the original configuration file are not preserved when writing the configuration back. What is considered a comment, depends on the given values for *comment_prefix* and *inline_comment_prefix*.

remove_option(*section*, *option*)

Remove the specified *option* from the specified *section*. If the section does not exist, raise `NoSectionError`. If the option existed to be removed, return `True`; otherwise return `False`.

remove_section(*section*)

Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)

Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before `optionxform()` is called.

configparser.**UNNAMED_SECTION**

A special object representing a section name used to reference the unnamed section (see *Unnamed Sections*).

configparser.**MAX_INTERPOLATION_DEPTH**

The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.11 RawConfigParser 物件

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

Legacy variant of the `ConfigParser`. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

在 3.2 版的變更: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

在 3.5 版的變更: 新增 *converters* 引數。

在 3.8 版的變更: The default *dict_type* is `dict`, since it now preserves insertion order.

在 3.13 版的變更: 新增 *allow_unnamed_section* 引數。

> **備🄵**
>
> Consider using *ConfigParser* instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

add_section(*section*)

Add a section named *section* or *UNNAMED_SECTION* to the instance.

If the given section already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised. If *UNNAMED_SECTION* is passed and support is disabled, *UnnamedSectionDisabledError* is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

在 3.14 版的變更: 新增對*UNNAMED_SECTION* 的支援。

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. While it is possible to use *RawConfigParser* (or *ConfigParser* with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.12 例外

exception configparser.**Error**

Base class for all other *configparser* exceptions.

exception configparser.**NoSectionError**

Exception raised when a specified section is not found.

exception configparser.**DuplicateSectionError**

Exception raised if *add_section()* is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

在 3.2 版的變更: Added the optional *source* and *lineno* attributes and parameters to `__init__()`.

exception configparser.**DuplicateOptionError**

Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.**NoOptionError**

Exception raised when a specified option is not found in the specified section.

exception configparser.**InterpolationError**

Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.**InterpolationDepthError**

Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception configparser.**InterpolationMissingOptionError**

Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception configparser.**InterpolationSyntaxError**

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception configparser.**MissingSectionHeaderError**

Exception raised when attempting to parse a file which has no section headers.

exception configparser.**ParsingError**

Exception raised when errors occur attempting to parse a file.

在 3.12 版的變更: The `filename` attribute and `__init__()` constructor argument were removed. They have been available using the name `source` since 3.2.

exception configparser.**MultilineContinuationError**

Exception raised when a key without a corresponding value is continued with an indented line.

在 3.13 版被加入.

exception configparser.**UnnamedSectionDisabledError**

Exception raised when attempting to use the *UNNAMED_SECTION* without enabling it.

在 3.14 版被加入.

exception configparser.**InvalidWriteError**

Exception raised when an attempted *ConfigParser.write()* would not be parsed accurately with a future *ConfigParser.read()* call.

Ex: Writing a key beginning with the *ConfigParser.SECTCRE* pattern would parse as a section header when read. Attempting to write this will raise this exception.

在 3.14 版被加入.

14.3 tomllib --- 剖析 TOML 檔案

在 3.11 版被加入.

原始碼： Lib/tomllib

此模組提供了剖析 TOML 1.0.0 (Tom's Obvious Minimal Language, https://toml.io) 的一個介面，此模組🄵不支援寫入 TOML。

也參考

Tomli-W 套件是一個 TOML 編寫器，可以與此模組結合使用，以提供標準函式庫中*marshal* 和*pickle* 模組之使用者所熟悉的寫入 API。

也參考

TOML 工具套件是一個保留風格且具有讀寫能力的 TOML 函式庫。若要編輯已存在的 TOML 文件，建議用它來替🄵此模組。

此模組定義了以下函式：

tomllib.**load**(*fp*, */*, ***, *parse_float=float*)

讀取一個 TOML 檔案。第一個引數應🄵一個可讀取的二進位檔案物件。回傳一個*dict*。用這個轉🄵表將 TOML 型🄵轉🄵成 Python 的。

parse_float 會被呼叫於要解碼的每個 TOML 浮點數字串。預設情🄵下，這相當於 `float(num_str)`。若有使用另一種資料型🄵或剖析器的 TOML 浮點數（例如*decimal.Decimal*），這就派得上用場。可呼叫物件不得回傳*dict* 或*list*，否則會引發*ValueError*。

不合格的 TOML 文件會使得*TOMLDecodeError* 被引發。

`tomllib.`**`loads`**(*s*, /, *, *parse_float=float*)

自一個*str* 物件載入成 TOML。回傳一個*dict*。用這個轉🄵表轉🄵 TOML 型🄵成 Python 的。*parse_float* 引數和*load()* 中的相同。

不合格的 TOML 文件會使得*TOMLDecodeError* 被引發。

以下🄵可用的例外：

exception `tomllib.`**`TOMLDecodeError`**(*msg*, *doc*, *pos*)

Subclass of *ValueError* with the following additional attributes:

msg

The unformatted error message.

doc

The TOML document being parsed.

pos

The index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

在 3.14 版的變更: Added the *msg*, *doc* and *pos* parameters. Added the *msg*, *doc*, *pos*, *lineno* and *colno* attributes.

在 3.14 版之後被🄵用: Passing free-form positional arguments is deprecated.

14.3.1 范例

剖析一個 TOML 檔案：

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

剖析一個 TOML 字串：

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

14.3.2 轉🄵表

TOML	Python
TOML 文件	dict
string	str
integer	int
float	float（可透過 *parse_float* 調整）
boolean	bool
偏移日期時間 (offset date-time)	datetime.datetime（設定 `tzinfo` 屬性🄵 `datetime.timezone` 的實例）
本地日期時間 (local date-time)	datetime.datetime（設定 `tzinfo` 🄵 `None`）
本地日期 (local date)	datetime.date
本地時間 (local time)	datetime.time
array	list
table	dict
行🄵表格 (inline table)	dict
表格陣列 (array of tables)	dict 串列 (list of dicts)

14.4 `netrc` --- netrc 檔案處理

原始碼： Lib/netrc.py

---

netrc 類🄵能🄵剖析 (parse) 🄵封裝 (encapsulate) netrc 檔案格式，以供 Unix **ftp** 程式和其他 FTP 用🄵端使用。

class netrc.**netrc**([*file*])

netrc 實例或其子類🄵實例能🄵封裝來自 netrc 檔案的資料。可用初始化引數（如有給定）指定要剖析的檔案，如果未給定引數，則將讀取（由*os.path.expanduser()* 指定的）使用者主目🄵中的 `.netrc` 檔案，否則將引發*FileNotFoundError* 例外。剖析錯誤會引發*NetrcParseError*，其帶有包括檔案名稱、列號和終止 token 的診斷資訊。

如果在 POSIX 系統上未指定引數，且若檔案所有權或權限不安全（擁有者與運行該行程的使用者不同，或者可供任何其他使用者讀取或寫入），存有密碼的 `.netrc` 檔案將會引發*NetrcParseError*。這實作了與 ftp 和其他使用 `.netrc` 程式等效的安全行🄵。這種安全檢查在不支援*os.getuid()* 的平台上不可用。

在 3.4 版的變更: 新增了 POSIX 權限檢查。

在 3.7 版的變更: 當未傳遞 *file* 引數時，*os.path.expanduser()* 可用於查找 `.netrc` 檔案的位置。

在 3.10 版的變更: *netrc* 在使用特定語言環境編碼前會先嘗試 UTF-8 編碼。netrc 檔案中的條目就不再需要包含所有 token，缺少的 token 值被預設🄵空字串。現在所有 token 及其值都可以包含任意字元，例如空格和非 ASCII 字元。如果登入名稱🄵匿名，就不會觸發安全檢查。

exception netrc.**NetrcParseError**

當原始文本中遇到語法錯誤時，*netrc* 類🄵會引發例外。此例外的實例提供了三個有趣的屬性:

msg

錯誤的文字解釋。

filename

原始檔案的名稱。

lineno

發現錯誤的列號。

14.4.1 netrc 物件

netrc 實例具有以下方法：

netrc.**authenticators**(*host*)

回傳 *host* 身份驗證器的三元素 tuple `(login, account, password)`。如果 netrc 檔案不包含給定主機的條目，則回傳與'default' 條目關聯的 tuple。如果🄵無匹配主機且預設條目也不可用則回傳 `None`。

netrc.**__repr__**()

將類🄵資料傾印 (dump) 🄵 netrc 檔案格式的字串。（這會將🄵解移除，🄵可能會對條目重新排序。）

netrc 的實例具有公開實例變數：

netrc.**hosts**

將主機名稱對映到 `(login, account, password)` tuple 的字典。'default' 條目（如存在）表示🄵該名稱對應到的🄵主機 (pseudo-host)。

netrc.**macros**

巨集 (macro) 名稱與字串 list（串列）的對映字典。

14.5 `plistlib` --- 🄵生和剖析 Apple `.plist` 檔案

原始碼： Lib/plistlib.py

This module provides an interface for reading and writing the "property list" files used by Apple, primarily on macOS and iOS. This module supports both binary and XML plist files.

The property list (`.plist`) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the *dump()* and *load()* functions.

To work with plist data in bytes or string objects, use *dumps()* and *loads()*.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), *bytes*, *bytearray* or *datetime.datetime* objects.

在 3.4 版的變更: New API, old API deprecated. Support for binary format plists added.

在 3.8 版的變更: Support added for reading and writing *UID* tokens in binary plists as used by NSKeyedArchiver and NSKeyedUnarchiver.

在 3.9 版的變更: Old API removed.

也參考

PList manual page
Apple's documentation of the file format.

此模組定義了以下函式：

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid:

- *None*: Autodetect the file format
- *FMT_XML*: XML file format
- *FMT_BINARY*: Binary plist format

The *dict_type* is the type used for dictionaries that are read from the plist file.

When *aware_datetime* is true, fields with type `datetime.datetime` will be created as *aware object*, with `tzinfo` as *datetime.UTC*.

XML data for the *FMT_XML* format is parsed using the Expat parser from *xml.parsers.expat* -- see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser raises *InvalidFileException* when the file cannot be parsed.

在 3.4 版被加入.

在 3.13 版的變更: 新增僅限關鍵字參數 *aware_datetime*。

plistlib.**loads**(*data*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Load a plist from a bytes or string object. See *load()* for an explanation of the keyword arguments.

在 3.4 版被加入.

在 3.13 版的變更: *data* 在 *fmt* 等於*FMT_XML* 時可以是一個字串。

plistlib.**dump**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Write *value* to a plist file. *fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values:

- *FMT_XML*: XML formatted plist file
- *FMT_BINARY*: Binary formatted plist file

When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises *TypeError* when a key of a dictionary is not a string, otherwise such keys are skipped.

When *aware_datetime* is true and any field with type `datetime.datetime` is set as an *aware object*, it will convert to UTC timezone before writing it.

A *TypeError* will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An *OverflowError* will be raised for integer values that cannot be represented in (binary) plist files.

在 3.4 版被加入.

在 3.13 版的變更: 新增僅限關鍵字參數 *aware_datetime*。

plistlib.**dumps**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Return *value* as a plist-formatted bytes object. See the documentation for *dump()* for an explanation of the keyword arguments of this function.

在 3.4 版被加入.

The following classes are available:

class plistlib.**UID**(*data*)

Wraps an *int*. This is used when reading or writing NSKeyedArchiver encoded data, which contains UID (see PList manual).

data

Int value of the UID. It must be in the range `0 <= data < 2**64`.

在 3.8 版被加入.

以下常數可供使用:

plistlib.**FMT_XML**

The XML format for plist files.

在 3.4 版被加入.

plistlib.**FMT_BINARY**

The binary format for plist files

在 3.4 版被加入.

這個模組定義了以下例外：

exception plistlib.**InvalidFileException**

當檔案無法被剖析時引發。

在 3.4 版被加入.

14.5.1 范例

Generating a plist:

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

Parsing a plist:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CHAPTER 15

加密服務

本章所描述的模組實作了多種加密演算法。它們可以在安裝時選擇是否一同安裝。以下🄵概述：

15.1 hashlib --- 安全雜🄵與訊息摘要

原始碼： Lib/hashlib.py

該模組實作了許多不同雜🄵演算法的通用介面，其中包括 FIPS 安全雜🄵演算法 SHA1、SHA224、SHA256、SHA384、SHA512（定義於 FIPS 180-4 標准）、SHA-3 系列（定義於 FIPS 202 標准）以及遺留的 SHA1 演算法（先前🄵 FIPS 的一部分）和 MD5 演算法（定義於網際網路 **RFC 1321**）。

> **備🄵**
>
> 如果你需要 adler32 或 crc32 雜🄵函式，可以在`zlib`模組中找到它們。

15.1.1 雜🄵演算法

每種種類的 *hash* 都有一個以其命名的建構函式方法。全部都會回傳具有相同簡單介面的雜🄵物件。例如：可使用`sha256()`來建立 SHA-256 雜🄵物件。現在你可以使用`update`方法向此物件提供*類位元組物件 (bytes-like objects)*（通常是`bytes`）。在任何時候，你都可以使用`digest()`或`hexdigest()`方法來要求它提供迄今🄵止傳遞給它的資料串聯的*摘要 (digest)*。

🄵了允許多執行緒 (multithreading)，Python *GIL* 被釋放，同時在其建構函式或`.update`方法中計算一次提供超過 2047 位元組資料的雜🄵值。

此模組中始終存在的雜🄵演算法之建構函式有`sha1()`、`sha224()`、`sha256()`、`sha384()`、`sha512()`、`sha3_224()`、`sha3_256()`、`sha3_384()`、`sha3_512()`、`shake_128()`、`shake_256()`、`blake2b()`和`blake2s()`。`md5()`通常也可用，但如果你使用罕見的「符合 FIPS (FIPS compliant)」的 Python 建置版本，它可能不存在或者不被允許使用。以上會對應到`algorithms_guaranteed`。

如果你的 Python 發行版的`hashlib`與提供其他演算法的 OpenSSL 版本鏈結，也可能有其他演算法可用。其他則不保證可用於在所有安裝上，🄵且只能以名稱來透過`new()`存取。請參🄵`algorithms_available`。

> **⚠ 警告**
>
> 某些演算法具有已知的雜🄵碰撞 (hash collision) 弱點（包括 MD5 和 SHA1）。請參🄵 Attacks on cryptographic hash algorithms 和本文件後面的*也參考*部分。

在 3.6 版被加入: 新增了 SHA3 (Keccak) 和 SHAKE 建構函式*sha3_224()*、*sha3_256()*、*sha3_384()*、*sha3_512()*、*shake_128()*、*shake_256()*。也新增了*blake2b()* 和*blake2s()*。 在 3.9 版的變更: 所有 hashlib 建構函式都🄵用一個僅限關鍵字引數 (keyword-only argument) *usedforsecurity*，其預設值🄵 `True`。False 值則會允許在受限環境中使用不安全 (insecure) 和阻塞的 (blocked) 雜🄵演算法。`False` 表示雜🄵演算法未在安全情境中使用，例如作🄵一種非加密用途的單向壓縮函式。

在 3.9 版的變更: Hashlib 現在使用 OpenSSL 中的 SHA3 和 SHAKE（如果有提供的話）。

在 3.12 版的變更: 對於鏈結之 OpenSSL 未提供的任何 MD5、SHA1、SHA2 或 SHA3 演算法，我們會回退使用 HACL* 專案中經過驗證的實作。

15.1.2 用法

取得位元組字串 `b"Nobody inspects the spammish repetition"` 的摘要:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0fK\
↪x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

更濃縮:

```
>>> hashlib.sha256(b"Nobody inspects the spammish repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

15.1.3 建構函式

`hashlib.`**`new`**(*name*, [*data*,]*, *usedforsecurity=True*)

是一個通用建構函式，它將目標演算法的字串 *name* 作🄵其第一個參數。它還允許使用者取得上面列出的雜🄵值以及 OpenSSL 函式庫可能提供的任何其他演算法。

使用*new()* 和演算法名稱:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

`hashlib.`**`md5`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha1`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha224`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha256`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha384`**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha512**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_512**([*data*,]*, *usedforsecurity=True*)

諸如此類的附名建構函式比將演算法名稱傳遞給*new()* 更快。

15.1.4 屬性

Hashlib 提供以下常數模組屬性：

hashlib.**algorithms_guaranteed**

包含所有平台上該模組保證支援的雜🄵演算法名稱的集合。請注意，'md5' 有出現在此列表中，🄵管有一些上游供應商提供了奇怪的「符合 FIPS (FIPS compliant)」的 Python 建置，🄵將其排除在外。

在 3.2 版被加入.

hashlib.**algorithms_available**

包含正在運行的 Python 直譯器中可用的雜🄵演算法名稱的集合。這些名稱在傳遞給*new()* 時將被識🄵。*algorithms_guaranteed* 都會是它的一個子集。相同的演算法可能會以不同的名稱多次出現在該集合中（多虧了 OpenSSL）。

在 3.2 版被加入.

15.1.5 雜🄵物件

以下的值皆🄵建構函式回傳之雜🄵物件的常數屬性：

hash.**digest_size**

生成雜🄵的大小（以位元組🄵單位）。

hash.**block_size**

雜🄵演算法的🄵部區塊大小（以位元組🄵單位）。

雜🄵物件具有以下屬性：

hash.**name**

該雜🄵的規範名稱，都會是小寫，且都會適合作🄵*new()* 的參數以建立此型🄵的另一個雜🄵。

在 3.4 版的變更: name 屬性自 CPython 誕生以來就存在於其中，但直到 Python 3.4 才正式確立，因此在某些平台上可能不存在。

雜🄵物件具有以下方法：

hash.**update**(*data*)

使用類位元組物件來更新雜🄵物件。重🄵呼叫相當於連接所有引數的單一呼叫：`m.update(a); m.update(b)` 等價於 `m.update(a+b)`。

hash.**digest**()

回傳目前有傳遞給*update()* 方法的資料之摘要。這是一個大小🄵*digest_size* 的位元組物件，它可能包含從 0 到 255 的整個範圍🄵的位元組。

hash.**hexdigest**()

與*digest()* 類似，只不過摘要會是作🄵雙倍長度的字串物件回傳，僅包含十六進位數字。這可用於在電子郵件或其他非二進位環境中安全地交🄵值。

hash.**copy**()

回傳雜🄵物件的副本（「🄵🄵 (clone)」），這可用於高效率地計算出共享同一初始子字串的資料之摘要。

15.1.6 SHAKE 可變長度摘要

hashlib.**shake_128**([*data*,]*, *usedforsecurity=True*)

hashlib.**shake_256**([*data*,]*, *usedforsecurity=True*)

shake_128() 和*shake_256()* 演算法提供了可變長度摘要，其 length_in_bits//2 最高🄵 128 或 256 位元安全性。因此，他們的摘要方法會需要長度。最大長度不受 SHAKE 演算法限制。

shake.**digest**(*length*)

回傳目前有傳遞給*update()* 方法的資料之摘要。這是一個大小🄵 *length* 的位元組物件，可能包含從 0 到 255 的整個範圍🄵的位元組。

shake.**hexdigest**(*length*)

與*digest()* 類似，只不過摘要作🄵雙倍長度的字串物件回傳，僅包含十六進位數字。這可用於交🄵電子郵件或其他非二進位環境中的值。

範例：

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

15.1.7 檔案雜🄵

hashlib 模組提供了一個輔助函式，用於對檔案或類檔案物件 (file-like object) 進行有效率的雜🄵。

hashlib.**file_digest**(*fileobj*, *digest*, */*)

回傳已用檔案物件🄵容更新的摘要物件。

fileobj 必須是以二進位模式讀取的類檔案物件。它接受來自🄵建*open()* 的檔案物件、*BytesIO* 實例、來自*socket.socket.makefile()* 的 SocketIO 物件等。*fileobj* 必須以阻塞模式開🄵，否則可能會引發*BlockingIOError*。

該函式可以繞過 Python 的 I/O 🄵直接使用*fileno()* 中的檔案描述器 (file descriptor)。在此函式回傳或引發後，必須假定 *fileobj* 處於未知狀態 (unknown state)。由呼叫者🄵定是否關閉 *fileobj*。

digest 必須是名稱作🄵 *str* 的雜🄵演算法、雜🄵建構函式或會回傳雜🄵物件的可呼叫函式。

範例：

```
>>> import io, hashlib, hmac
>>> with open("library/hashlib.rst", "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'
```

```
>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)
```

```
>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata", digestmod=hashlib.sha512)
>>> mac1.digest() == mac2.digest()
True
```

在 3.11 版被加入.

在 3.14 版的變更: Now raises a *BlockingIOError* if the file is opened in blocking mode. Previously, spurious null bytes were added to the digest.

15.1.8 密鑰的生成

密鑰生成 (key derivation) 和密鑰延伸 (key stretching) 演算法專🄵安全密碼雜🄵而設計。像是 `sha1(password)` 這樣過於單純的演算法無法抵抗暴力攻擊。一個好的密碼雜🄵函式必須是可調校的 (tunable)、緩慢的，🄵且有包含 salt（鹽）。

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

該函式提供基於密碼的 PKCS#5 密鑰生成函式 2。它使用 HMAC 作🄵🄵隨機函式 (pseudorandom function)。

字串 *hash_name* 是 HMAC 的雜🄵摘要演算法所需的名稱，例如 `sha1` 或 `sha256`。*password* 和 *salt* 被直譯🄵位元組緩衝區。應用程式和函式庫應🄵 *password* 設下合理的長度限制（例如 1024）。*salt* 應該是來自適當來源（例如*`os.urandom()`*）且大約 16 或更多位元組。

應根據雜🄵演算法和計算能力選擇 *iterations* 次數。截至 2022 年，建議進行數十萬次 SHA-256 🄵代。有關🄵什🄵以及如何選擇最適合你的應用程式的基本原理，請🄵讀 NIST-SP-800-132 的 *Appendix A.2.2*。stackexchange pbkdf2 🄵代問題上的答案有詳細解釋。

dklen 是生成的密鑰長度。如果 *dklen* 🄵 `None`，則會使用雜🄵演算法 *hash_name* 的摘要大小，例如 SHA-512 🄵 64。

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

僅當有使用 OpenSSL 編譯 Python 時該函式才可用。

在 3.4 版被加入.

在 3.12 版的變更: 該函式現在僅在有使用 OpenSSL 建置 Python 時可用。緩慢的純 Python 實作已被🄵除。

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

該函式提供（如 **RFC 7914** 中所定義的）scrypt 基於密碼的密鑰衍生函式。

password 和 *salt* 必須是類位元組物件。應用程式和函式庫應🄵 *password* 設下合理的長度限制（例如 1024）。*salt* 應該是來自適當來源（例如*`os.urandom()`*）且大約 16 或更多位元組。

n 是 CPU/記憶體開銷🄵數、*r* 是區塊大小、*p* 🄵平行化🄵數、*maxmem* 🄵記憶體限制（OpenSSL 1.1.0 預設🄵 32 MiB）。*dklen* 是生成密鑰的長度。

在 3.6 版被加入.

15.1.9 BLAKE2

BLAKE2 是在 **RFC 7693** 中定義的加密雜🄵函式，有兩種類型：

- **BLAKE2b**，針對 64 位元平台進行了最佳化，可生成 1 到 64 位元組之間任意大小的摘要，
- **BLAKE2s**，針對 8 至 32 位元平台進行了最佳化，可生成 1 至 32 位元組之間任意大小的摘要。

BLAKE2 支援**密鑰模式 (keyed mode)**（更快、更簡單的 HMAC 替代品）、**加鹽雜🄵 (salted hashing)**、**個人化**和**樹狀雜🄵**。

該模組中的雜🄵物件遵循標準函式庫的*`hashlib`* 物件 API。

建立雜🄵物件

新的雜🄵物件是透過呼叫建構函式建立的：

`hashlib.`**`blake2b`**(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

hashlib.**blake2s**(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

這些函式回傳相應的雜🄵物件以計算 BLAKE2b 或 BLAKE2s。他們可以選擇🄵用這些通用參數：

- *data*：要雜🄵的初始資料塊 (data chunk)，它必須是類位元組物件。它只能作🄵位置引數傳遞。
- *digest_size*：輸出摘要的大小（以位元組🄵單位）。
- *key*：用於密鑰雜🄵的密鑰（BLAKE2b 最多 64 位元組、BLAKE2s 最多 32 位元組）。
- *salt*：用於隨機雜🄵的鹽（BLAKE2b 最多 16 個位元組、BLAKE2s 最多 8 個位元組）。
- *person*：個人化字串（BLAKE2b 最多 16 個位元組、BLAKE2s 最多 8 個位元組）。

下表顯示了一般參數的限制（以位元組🄵單位）：

雜🄵	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

備🄵

BLAKE2 規範定義了鹽和個人化參數的🄵定長度，但🄵了方便起見，此實作接受任意大小的位元組字串，最多可達指定長度。如果參數的長度小於指定的長度，則用零補滿，所以像是 `b'salt` 和 `b'salt\x00` 是相同的值。（但 *key* 的情🄵🄵非如此。）

這些大小可作🄵模組常數使用，如下所述。

建構函式還接受以下樹狀雜🄵參數：

- *fanout*：扇出（0 到 255，如果無限制則🄵 0、順序模式🄵 1）。
- *depth*：樹的最大深度（1 到 255，如果無限制則🄵 255、順序模式🄵 1）。
- *leaf_size*：葉的最大位元組長度（0 到 `2**32-1`，如果無限制或處於順序模式則🄵 0）。
- *node_offset*：節點偏移量（BLAKE2b 🄵 0 到 `2**64-1`，BLAKE2s 🄵 0 到 `2**48-1`，0 表示第一個、最左邊、葉子或在順序模式下）。
- *node_depth*：節點深度（0 到 255，葉🄵 0，或在順序模式下）。
- *inner_size*：🄵部摘要大小（BLAKE2b 🄵 0 到 64，BLAKE2s 🄵 0 到 32，順序模式🄵 0）。
- *last_node*：布林值，代表處理的節點是否🄵最後一個（`False` 代表順序模式）。

關於樹狀雜🄵的綜合回顧，請參🄵 BLAKE2 規範中的第 2.10 節。

常數

blake2b.**SALT_SIZE**

blake2s.**SALT_SIZE**

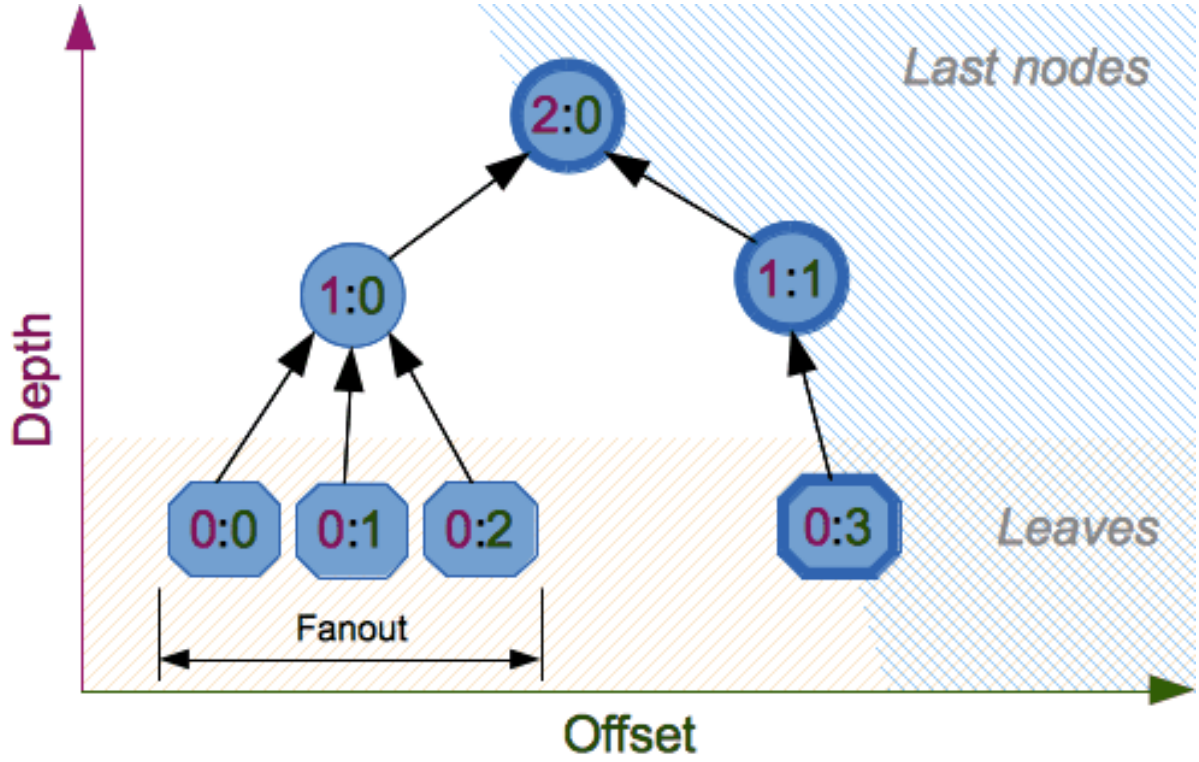
鹽長度（建構函式接受的最大長度）。

blake2b.**PERSON_SIZE**

blake2s.**PERSON_SIZE**

個人化字串長度（建構函式接受的最大長度）。

blake2b.**MAX_KEY_SIZE**



blake2s.**MAX_KEY_SIZE**

最大密鑰大小。

blake2b.**MAX_DIGEST_SIZE**

blake2s.**MAX_DIGEST_SIZE**

雜🄵函式可以輸出的最大摘要大小。

范例

簡單雜🄵

要計算某些資料的雜🄵值，你應該首先透過呼叫適當的建構函式（*blake2b()* 或*blake2s()*）以建構一個雜🄵物件，然後透過於物件呼叫*update()* 來以資料對它更新，最後透過呼叫*digest()*（或對於十六進位編碼字串則🄵*hexdigest()*）從物件中獲得摘要。

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

作🄵一個快捷方式，你可以將要更新的第一個資料塊作🄵位置引數直接傳遞給建構函式：

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

你可以根據需求來多次呼叫*hash.update()* 以🄵代更新雜🄵：

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
...
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

使用不同的摘要大小

BLAKE2 有可調整的摘要大小，BLAKE2b 最多🄵 64 位元組，BLAKE2s 最多🄵 32 位元組。例如，要在不改變輸出大小的情🄵下用 BLAKE2b 替🄵 SHA-1，我們可以指定 BLAKE2b 生成 20 位元組的摘要：

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

具有不同摘要大小的雜🄴物件具有完全不同的輸出（較短的雜🄴值不是較長雜🄴值的前綴）；即使輸出長度相同，BLAKE2b 和 BLAKE2s 也會🄵生不同的輸出：

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

密鑰雜🄵 (Keyed hashing)

密鑰雜🄵可用於身份驗證，作🄵基於雜🄴的訊息驗證碼 (Hash-based message authentication code) (HMAC) 的更快、更簡單的替代方案。由於繼承自 BLAKE 的不可微特性 (indifferentiability property)，BLAKE2 可以安全地用於 prefix-MAC 模式。

此範例示範了如何使用密鑰 `b'pseudorandom key'` 取得訊息 `b'message data'` 的（十六進位編碼）128 位元驗證碼：

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

舉一個實際的例子，網頁應用程式可以對發送給使用者的 cookie 進行對稱簽名 (symmetrically sign)，然後驗證它們以確保它們🄴有被篡改：

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
```

(繼續下一頁)

(繼續上一頁)

```
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

🄵管有原生密鑰雜🄵模式，BLAKE2 還是可以透過 *hmac* 模組用於建構 HMAC：

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

隨機雜🄵 (Randomized hashing)

透過設定 *salt* 參數，使用者可以向雜🄵函式引入隨機化。隨機雜🄵在防止針對數位簽章中雜🄵函式的碰撞攻擊 (collision attacks) 非常有用。

> 隨機雜🄵是🄵這樣的情🄵而設計的：一方（訊息准備者）生成全部或部分訊息🄵由另一方（訊息簽名者）簽名。如果訊息准備者能🄵發現加密雜🄵函式發生碰撞（collision，即兩條訊息🄵生相同的雜🄵值），那🄵他們可能會准備有意義的訊息版本，該版本將🄵生相同的雜🄵值和數位簽章，但結果不同（例如，將 $1,000,000 轉入賬🄵，而不是 $10）。加密雜🄵函式的設計以抗碰撞性🄵主要目標，但目前對加密雜🄵函式攻擊的關注可能會導致給定的加密雜🄵函式所提供的抗碰撞性低於預期。隨機雜🄵透過降低准備者在數位簽章生成過程中生成最終🄵生相同雜🄵值的兩個或多個訊息的可能性，🄵簽名者提供額外的保護——即便嘗試去找到雜🄵函式碰撞的發生是實際可行的。然而，若訊息的所有部分都是由簽名者所准備好的，使用隨機雜🄵可能會降低數位簽章提供的安全性。
>
> （NIST SP-800-106「數位簽章的隨機雜🄵 (Randomized Hashing for Digital Signatures)」）

在 BLAKE2 中，鹽在初始化期間作🄵雜🄵函式的一次性輸入被處理，而不是作🄵每個壓縮函式的輸入。

> **⚠ 警告**
>
> 使用 BLAKE2 或任何其他通用加密雜🄵函式（例如 SHA-256）的加鹽雜🄵（或單純雜🄵）不適合對密碼進行雜🄵處理。有關更多資訊，請參🄵 BLAKE2 FAQ 。

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
```

(繼續下一頁)

(繼續上一頁)

```
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

個人化

有時候🄵制雜🄵函式🄵不同目的的相同輸入生成不同的摘要是很有用的。引用 Skein 雜🄵函式的作者的話：

> 我們建議所有應用程式設計者認真考慮這樣做；我們已經看到許多協定，其中在協定的一個部分中計算的雜🄵可以在完全不同的部分中使用，因🄵兩次雜🄵計算是在相似或相關的資料上完成的，🄵且攻擊者可以🄵制應用程式將雜🄵輸入設🄵相同的。對協定中使用的每個雜🄵函式進行個人化可以立即阻止此類攻擊。
>
> （Skein 雜🄵函式系列，第 21 頁）

BLAKE2 可以透過將位元組傳遞給 *person* 引數來做個人化：

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

個人化與密鑰模式還可以一起用於從單個密鑰得出不同的密鑰。

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

樹狀模式

下面是對具有兩個葉節點的最小樹進行雜🄵處理的範例：

```
  10
 /  \
00  01
```

此範例使用 64-byte 🄵部摘要，🄵回傳 32-byte 最終摘要：

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

作人員

BLAKE2 由 *Jean-Philippe Aumasson*、*Samuel Neves*、*Zooko Wilcox-O'Hearn* 和 *Christian Winnerlein* 設計，基於由 *Jean-Philippe Aumasson*、*Luca Henzen*、*Willi Meier* 和 *Raphael C.-W. Phan* 所建立的 SHA-3 最終版本 BLAKE。

它使用 *Daniel J. Bernstein* 設計的 ChaCha 密碼的核心演算法。

標准函式庫實作是基於 pyblake2 模組。它是由 *Dmitry Chestnykh* 在 *Samuel Neves* 的 C 版本實作F基礎所編寫的。該文件是由 *Dmitry Chestnykh* 編寫F從 pyblake2 FF過來的。

Christian Heimes F Python 重寫了部分 C 程式碼。

以下公開領域貢獻適用於 C 雜F函式實作、擴充程式碼和此文件：

> 在法律允許的範圍F，作者已將該軟體的所有版權以及相關和鄰接權利奉獻給全球的公開領域。該軟體的發布F有任何授權 (warranty)。
>
> 你應會隨本軟體一起收到一份 CC0 公F領域貢獻宣告 (CC0 Public Domain Dedication) 的副本。如果F有，請參F https://creativecommons.org/publicdomain/zero/1.0/ 。

以下人員根據創用 CC 通用公F領域貢獻宣告 1.0 (Creative Commons Public Domain Dedication 1.0 Universal) 於專案和公開領域做出了開發或貢獻：

- *Alexandr Sokolovskiy*

也參考

`hmac` 模組
使用雜F生成訊息驗證程式碼的模組。

`base64` 模組
另一種在非二進位環境中編碼二進位雜F的方法。

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
有關安全雜F演算法的 FIPS 180-4 出版物。

https://csrc.nist.gov/pubs/fips/202/final
有關 SHA-3 標准的 FIPS 202 出版物。

https://www.blake2.net/
BLAKE2 官方網站。

https://en.wikipedia.org/wiki/Cryptographic_hash_function
包含有關哪些演算法存在已知問題以及這些問題對其使用意味著什F資訊的維基百科文章。

https://www.ietf.org/rfc/rfc8018.txt
PKCS #5：基於密碼的加密規範版本 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
NIST（美國國家標准技術研究院）針對基於密碼的密鑰衍生的建議。

15.2 hmac --- 基於金鑰雜F的訊息驗証

原始碼： Lib/hmac.py

此模組實作了 **RFC 2014** 所描述的 HMAC 演算法。此介面允許使用具有固定摘要 (digest) 長度的任意雜F函式。要特F注意的是擴充輸出函式（如 SHAKE-128 或 SHAKE-256）不能與 HMAC 一起使用。

hmac.**new**(*key*, *msg=None*, *digestmod*)

回傳一個新的 hmac 物件。*key* 是一個指定密鑰的 bytes（位元組）或 bytearray 物件。如果提供了 *msg*，將會呼叫 `update(msg)` 方法。*digestmod* F HMAC 物件所用的摘要名稱、摘要建構函式 (constructor) 或模組。它可以是適用於*hashlib.new()* 的任何名稱。F管該引數的位置在後，但它F是必須的。

在 3.4 版的變更: 參數 *key* 可以F bytes 或 bytearray 物件。參數 *msg* 可以F*hashlib* 所支援的任意型F。參數 *digestmod* 可以F雜F演算法的名稱。

在 3.8 版的變更: *digestmod* 引數現在是必須的。請將其作F關鍵字引數傳入以避免當你F有初始 *msg* 時導致的麻煩。

hmac.**digest**(*key*, *msg*, *digest*)

基於給定密鑰 *key* 和 *digest* 回傳 *msg* 的摘要。此函式等價於 `HMAC(key, msg, digest).digest()`，但使用了優化的 C 或行F實作（inline implementation），對放入記憶體的訊息能處理得更快。參數 *key*、*msg* 和 *digest* 在*new()* 中具有相同含義。

作F CPython 的實現細節，C 的優化實作只有當 *digest* F字串F且是一個 OpenSSL 所支持的摘要演算法的名稱時才會被使用。

在 3.7 版被加入.

class hmac.**HMAC**

HMAC 物件具有下列方法 (method)：

HMAC.**update**(*msg*)

用 *msg* 來更新 hmac 物件。重F呼叫相當於單次呼叫F傳入所有引數的拼接結果：`m.update(a); m.update(b)` 等價於 `m.update(a + b)`。

在 3.4 版的變更: 參數 *msg* 可以是*hashlib* 所支援的任何型F。

HMAC.**digest**()

回傳目前已傳給*update()* 方法的 bytes 摘要。這個 bytes 物件的長度會與傳給建構函式的摘要 *digest_size* 的長度相同。它可以包含 NUL bytes 以及 non-ASCII bytes。

> **警告**
>
> 在一個例行的驗證事務運行期間，將*digest()* 的輸出與外部提供的摘要進行比較時，建議使用*compare_digest()* 函式而不是 == 運算子以F少被定時攻擊時的漏洞。

HMAC.**hexdigest**()

像是*digest()* 但摘要的回傳形式🄵兩倍長度的字串，且此字串只包含十六進位數位。這可以被用於在電子郵件或其他非二進位制環境中安全地交🄵數據。

> **警告**
>
> 在一個例行的驗證事務運行期間，將*hexdigest()* 的輸出與外部提供的摘要進行比較時，建議使用*compare_digest()* 函式而不是 == 運算子以🄵少被定時攻擊時的漏洞。

HMAC.**copy**()

回傳 hmac 物件的拷貝 (”clone”)。這可以被用來有效率地計算那些共享相同初始子字串的字串的摘要。

一個 hash 物件具有以下屬性：

HMAC.**digest_size**

以 bytes 表示最終 HMAC 摘要的大小。

HMAC.**block_size**

以 bytes 表示雜🄴演算法的🄵部區塊大小。

在 3.4 版被加入.

HMAC.**name**

HMAC 的正准名稱總是🄵小寫形式，例如 `hmac-md5`。

在 3.4 版被加入.

在 3.10 版的變更: 未寫入文件的屬性 `HMAC.digest_cons`，`HMAC.inner` 和 `HMAC.outer` 已被移除。

這個模組還提供了下列輔助函式：

hmac.**compare_digest**(*a*, *b*)

回傳 `a == b`。此函式使用一種經專門設計的方式通過避免基於🄵容的短路行🄵來防止定時分析，使得它適合處理密碼學。*a* 和 *b* 必須🄵相同的型🄵：可以是*str* (僅限 ASCII，如*HMAC.hexdigest()* 的回傳值)，或者是*bytes-like object*。

> **備🄵**
>
> 如果 *a* 和 *b* 具有不同的長度，或者如果發生了錯誤，定時攻擊在理論上可以取得有關 *a* 和 *b* 的型🄵和長度的訊息—但不能取得他們的值。

在 3.3 版被加入.

在 3.10 版的變更: 此函式在可能的情🄵下會在🄵部使用 OpenSSL 的 `CRYPTO_memcmp()`。

> **也參考**
>
> ***hashlib* 模組**
> Python 模組提供安全的雜🄴函式。

15.3 `secrets` --- 🄵生用於管理機密的安全亂數

在 3.6 版被加入.

原始碼： Lib/secrets.py

secrets 模組可用於🄵生高加密🄵度的亂數，適合用來管理諸如密碼、帳號認證、安全性權杖（security tokens）這類資料，以及管理其他相關的機密資料。

尤其應優先使用*secrets* 作🄵預設來替代*random* 模組中的預設🄵亂數🄵生器（pseudo-random number generator），該模組被設計用於建模和模擬，而非用於安全性和加密。

也參考

PEP 506

15.3.1 亂數

secrets 模組使你得以存取作業系統所提供安全性最高的亂數🄵生器。

class secrets.**SystemRandom**

一個用來🄵生亂數的類🄵，用的是作業系統提供的最高品質來源。請參🄵*random.SystemRandom* 以取得更多細節。

secrets.**choice**(*seq*)

從一非空序列中，回傳一個隨機選取的元素。

secrets.**randbelow**(*exclusive_upper_bound*)

回傳一個 [0, *exclusive_upper_bound*) 範圍之🄵的隨機整數。

secrets.**randbits**(*k*)

回傳一個具 *k* 個隨機位元的非負整數。

15.3.2 🄵生權杖（token）

secrets 模組提供了一些🄵生安全性權杖的函式，適合用於諸如重設密碼、難以猜測的 URL，或類似的應用。

secrets.**token_bytes**([*nbytes=None*])

回傳一個隨機位元組字串，其中含有 *nbytes* 位元組的數字。如果 *nbytes* 🄵 None 或未提供，則會使用一合理預設值。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

回傳一以十六進位表示的隨機字串。字串具有 *nbytes* 個隨機位元組，每個位元組會轉成兩個十六進位的數字。如果 *nbytes* 🄵 None 或未提供，則會使用一個合理的預設值。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

回傳一個 URL 安全的隨機文本字串，包含 *nbytes* 個隨機位元組。文本將使用 Base64 編碼，因此平均來🄵每個位元組會對應到約 1.3 個字元。如果 *nbytes* 🄵 None 或未提供，則會使用一個合理的預設值。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

權杖應當使用多少個位元組？

🄵了在面對暴力攻擊時能保證安全，權杖必須具有足🄵的隨機性。不幸的是，對隨機性是否足🄵的標準，會隨著電腦越來越🄵大🄵能🄵在更短時間🄵進行更多猜測而不斷提高。在 2015 年時，人們認🄵 32 位元組（256 位元）的隨機性對於*secrets* 模組所預期的一般使用場景來🄵是足🄵的。

對於想自行管理權杖長度的使用者，你可以對各種 `token_*` 函式明白地指定*int* 引數（argument）來指定權杖要使用的隨機性程度。該引數以位元組數來表示要使用的隨機性程度。

否則，如未提供引數，或者如果引數🄵 `None`，則 `token_*` 函式則會使用一個合理的預設值。

> **備🄵**
>
> 該預設值可能在任何時候被改變，包括在維護版本更新的時候。

15.3.3 其他函式

`secrets.`**`compare_digest`**(*a*, *b*)

如果字串或類位元組串物件 *a* 與 *b* 相等則回傳 `True`，否則回傳 `False`，以”🄵定時間比較 (constant-time compare) ” 的處理方式可降低時序攻擊的風險。請參🄵*`hmac.compare_digest()`* 以了解更多細節。

15.3.4 應用技巧和典范實務（best practices）

本節展示了一些使用*`secrets`* 來管理基本安全等級的應用技巧和典範實務。

🄵生八個字元長的字母數字密碼：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

> **備🄵**
>
> 應用程式不能**以可復原的格式儲存密碼**，無論是用純文本還是經過加密。它們應當先加鹽（salt），再使用高加密🄴度的單向（不可逆）雜🄵函式來🄵生雜🄵值。

🄵生十個字元長的字母數字密碼，其中包含至少一個小寫字母，至少一個大寫字母以及至少三個數字：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

🄵生 XKCD 風格的 passphrase：

```
import secrets
# 在標准 Linux 系統上，使用一個方便的字典檔案。
# 其他平台可能需要提供自己的字詞列表。
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

🄵生難以猜測的暫時性 URL，🄵含回復密碼時所用的一個安全性權杖：

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPTER 16

通用作業系統服務

此章節所描述的模組 (module) 提供了作業系統特性的使用介面，例如檔案與時鐘，（幾乎）在所有作業系統上皆能使用。這些介面通常是參考 Unix 或 C 的介面來實作，不過在其他大多數系統上也能使用。以下🄵概述：

16.1 os --- 各種作業系統介面

原始碼： Lib/os.py

該模組提供了一種便利的方式來操作與作業系統相關的功能。如果你想讀取或寫入檔案，請參🄵*open()*，如果你想操作檔案路徑，請參🄵*os.path* 模組，如果你想透過命令列查看所有檔案中的所有🄵容，請查看*fileinput* 模組。要建立臨時檔案和目🄵，請參🄵*tempfile* 模組，要操作高級檔案和目🄵，請參🄵*shutil* 模組。

關於這些功能的可用性🄵明：

- Python 所有🄵建作業系統相關的模組設計是這樣：只要有相同的函式可使用，就會使用相同的介面（interface）。舉例來🄵，`os.stat(path)` 函式會以相同格式回傳關於 *path* 的統計資訊（這剛好來自於 POSIX 的介面。）。
- 對於特定的作業系統獨有的擴充功能也可以透過*os* 取得，但使用它們的時候對於可移植性無疑會是個問題。
- 所有接受檔案路徑和檔案名稱的函式皆接受位元組 (bytes) 和字串物件 (string objects)，且如果回傳檔案路徑或檔案名稱，則會🄵出相同型🄵的物件。
- 在 VxWorks, 不支援 os.popen、os.fork、os.execv 和 os.spawn*p*。
- 在 WebAssembly 平台和 Android 與 iOS 上，大部分*os* 模組無法使用或行🄵不同。與行程 (process)（例如*fork()*、*execve()*）與資源（例如*nice()*）相關的 API 不可使用。其他諸如*getuid()* 和*getpid()* 的相關 API 是 emulated 或 stubs。WebAssembly 平台也缺少訊號相關支援（例如*kill()*、*wait()*）。

備🄵

在檔案名稱和路徑找不到或無效的時候，或引數型🄵正確但作業系統不接受的時候，在此模組中的所有的函式都會引發*OSError*（或其子類🄵）。

exception os.**error**

🄵建例外*OSError* 的🄵名。

os.**name**

The name of the operating system dependent module imported. The following names have currently been registered: `'posix'`, `'nt'`, `'java'`.

> **也參考**
>
> *sys.platform* has a finer granularity. *os.uname()* gives system-dependent version information.
>
> The *platform* module provides detailed checks for the system's identity.

16.1.1 File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the *filesystem encoding and error handler* to perform this conversion (see *sys.getfilesystemencoding()*).

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

在 3.1 版的變更: On some systems, conversion using the file system encoding may fail. In this case, Python uses the *surrogateescape encoding error handler*, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The *file system encoding* must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise *UnicodeError*.

另請參🄵*locale encoding*。

16.1.2 Python UTF-8 模式

在 3.7 版被加入: 更多資訊請見 **PEP 540**。

Python 在 UTF-8 模式下會忽略*locale encoding* 🄵且🄵制使用 UTF-8 去編碼:

- Use UTF-8 as the *filesystem encoding*.
- *sys.getfilesystemencoding()* 回傳 `'utf-8'`。
- *locale.getpreferredencoding()* 回傳 `'utf-8'`（*do_setlocale* 引數🄵無任何效果）。
- *sys.stdin*, *sys.stdout*, and *sys.stderr* all use UTF-8 as their text encoding, with the `surrogateescape` *error handler* being enabled for *sys.stdin* and *sys.stdout* (*sys.stderr* continues to use `backslashreplace` as it does in the default locale-aware mode)
- 在 Unix 中，*os.device_encoding()* 回傳 `'utf-8'` 而非裝置編碼。

請注意，在 UTF-8 模式中，標準的串流設定會被 `PYTHONIOENCODING` 覆蓋（如同在預設在地化覺察模式一樣）。

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- 命令列引數、環境變數及檔案名稱會以 UTF-8 編碼形式來被解碼成文字。
- *os.fsdecode()* 和*os.fsencode()* 使用 UTF-8 編碼。
- *open()*, *io.open()*, and *codecs.open()* use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The *Python UTF-8 Mode* is enabled if the LC_CTYPE locale is `C` or `POSIX` at Python startup (see the `PyConfig_Read()` function).

It can be enabled or disabled using the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

If the `PYTHONUTF8` environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from *sys.flags.utf8_mode*.

See also the UTF-8 mode on Windows and the *filesystem encoding and error handler*.

也參考

PEP 686
Python 3.15 預設使用*Python UTF-8 模式*

16.1.3 行程參數

These functions and data items provide information and operate on the current process and user.

`os.`**`ctermid`**`()`

Return the filename corresponding to the controlling terminal of the process.

可用性: Unix, not WASI.

`os.`**`environ`**

A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the *os* module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in *os.environ*, except for changes made by modifying *os.environ* directly.

This mapping may be used to modify the environment as well as query the environment. *putenv()* will be called automatically when the mapping is modified.

On Unix, keys and values use *sys.getfilesystemencoding()* and `'surrogateescape'` error handler. Use *environb* if you would like to use a different encoding.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

備🄵

呼叫*putenv()* 🄵不會改變*os.environ*，所以建議直接修改*os.environ*。

備🄵

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

You can delete items in this mapping to unset environment variables. *unsetenv()* will be called automatically when an item is deleted from *os.environ*, and when one of the `pop()` or `clear()` methods is called.

> **也參考**
>
> `os.reload_environ()` 函式。

在 3.9 版的變更: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

`os.`**`environb`**

Bytes version of `environ`: a *mapping* object where both keys and values are `bytes` objects representing the process environment. `environ` and `environb` are synchronized (modifying `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is `True`.

在 3.2 版被加入.

在 3.9 版的變更: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

`os.`**`reload_environ`**`()`

The `os.environ` and `os.environb` mappings are a cache of environment variables at the time that Python started. As such, changes to the current process environment are not reflected if made outside Python, or by `os.putenv()` or `os.unsetenv()`. Use `os.reload_environ()` to update `os.environ` and `os.environb` with any such changes to the current process environment.

> **警告**
>
> This function is not thread-safe. Calling it while the environment is being modified in an other thread is an undefined behavior. Reading from `os.environ` or `os.environb`, or calling `os.getenv()` while reloading, may return an empty result.

在 3.14 版被加入.

`os.`**`chdir`**`(`*path*`)`

`os.`**`fchdir`**`(`*fd*`)`

`os.`**`getcwd`**`()`

These functions are described in *Files and Directories*.

`os.`**`fsencode`**`(`*filename*`)`

Encode *path-like* *filename* to the *filesystem encoding and error handler*; return `bytes` unchanged.

`fsdecode()` is the reverse function.

在 3.2 版被加入.

在 3.6 版的變更: Support added to accept objects implementing the `os.PathLike` interface.

`os.`**`fsdecode`**`(`*filename*`)`

Decode the *path-like* *filename* from the *filesystem encoding and error handler*; return `str` unchanged.

`fsencode()` is the reverse function.

在 3.2 版被加入.

在 3.6 版的變更: Support added to accept objects implementing the `os.PathLike` interface.

`os.`**`fspath`**`(`*path*`)`

Return the file system representation of the path.

If `str` or `bytes` is passed in, it is returned unchanged. Otherwise `__fspath__()` is called and its value is returned as long as it is a `str` or `bytes` object. In all other cases, `TypeError` is raised.

在 3.6 版被加入.

class os.**PathLike**

An *abstract base class* for objects representing a file system path, e.g. *pathlib.PurePath*.

在 3.6 版被加入.

abstractmethod **__fspath__**()

Return the file system path representation of the object.

The method should only return a *str* or *bytes* object, with the preference being for *str*.

os.**getenv**(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn't. *key* is a string. Note that since *getenv()* uses *os.environ*, the mapping of *getenv()* is similarly also captured on import, and the function may not reflect future environment changes.

On Unix, keys and values are decoded with *sys.getfilesystemencoding()* and `'surrogateescape'` error handler. Use *os.getenvb()* if you would like to use a different encoding.

可用性: Unix, Windows.

os.**getenvb**(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since *getenvb()* uses *os.environb*, the mapping of *getenvb()* is similarly also captured on import, and the function may not reflect future environment changes.

getenvb() is only available if *supports_bytes_environ* is `True`.

可用性: Unix.

在 3.2 版被加入.

os.**get_exec_path**(*env=None*)

Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. *env*, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when *env* is `None`, *environ* is used.

在 3.2 版被加入.

os.**getegid**()

Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

可用性: Unix, not WASI.

os.**geteuid**()

Return the current process's effective user id.

可用性: Unix, not WASI.

os.**getgid**()

Return the real group id of the current process.

可用性: Unix.

The function is a stub on WASI, see *WebAssembly 平台* for more information.

os.**getgrouplist**(*user*, *group*, */*)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.**getgroups**()

Return list of supplemental group ids associated with the current process.

可用性: Unix, not WASI.

> **備🄵**
>
> On macOS, *getgroups()* behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, *getgroups()* returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to *setgroups()* if suitably privileged. If built with a deployment target greater than `10.5`, *getgroups()* returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to *setgroups()*, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with *sysconfig.get_config_var()*.

os.**getlogin**()

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use *getpass.getuser()* since the latter checks the environment variables `LOGNAME` or `USERNAME` to find out who the user is, and falls back to `pwd.getpwuid(os.getuid())[0]` to get the login name of the current real user id.

可用性: Unix, Windows, not WASI.

os.**getpgid**(*pid*)

Return the process group id of the process with process id *pid*. If *pid* is 0, the process group id of the current process is returned.

可用性: Unix, not WASI.

os.**getpgrp**()

Return the id of the current process group.

可用性: Unix, not WASI.

os.**getpid**()

Return the current process id.

The function is a stub on WASI, see *WebAssembly 平台* for more information.

os.**getppid**()

Return the parent's process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

可用性: Unix, Windows, not WASI.

在 3.2 版的變更: 新增對 Windows 的支援。

os.**getpriority**(*which*, *who*)

Get program scheduling priority. The value *which* is one of *PRIO_PROCESS*, *PRIO_PGRP*, or *PRIO_USER*, and *who* is interpreted relative to *which* (a process identifier for *PRIO_PROCESS*, process group identifier for *PRIO_PGRP*, and a user ID for *PRIO_USER*). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.**PRIO_PROCESS**

os.**PRIO_PGRP**

os.**PRIO_USER**

Parameters for the *getpriority()* and *setpriority()* functions.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.**PRIO_DARWIN_THREAD**

os.**PRIO_DARWIN_PROCESS**

os.**PRIO_DARWIN_BG**

os.**PRIO_DARWIN_NONUI**

Parameters for the *getpriority()* and *setpriority()* functions.

可用性: macOS

在 3.12 版被加入.

os.**getresuid**()

Return a tuple (ruid, euid, suid) denoting the current process's real, effective, and saved user ids.

可用性: Unix, not WASI.

在 3.2 版被加入.

os.**getresgid**()

Return a tuple (rgid, egid, sgid) denoting the current process's real, effective, and saved group ids.

可用性: Unix, not WASI.

在 3.2 版被加入.

os.**getuid**()

Return the current process's real user id.

可用性: Unix.

The function is a stub on WASI, see *WebAssembly 平台* for more information.

os.**initgroups**(*username*, *gid*, /)

Call the system initgroups() to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

可用性: Unix, not WASI, not Android.

在 3.2 版被加入.

os.**putenv**(*key*, *value*, /)

Set the environment variable named *key* to the string *value*. Such changes to the environment affect subprocesses started with *os.system()*, *popen()* or *fork()* and *execv()*.

Assignments to items in *os.environ* are automatically translated into corresponding calls to *putenv()*; however, calls to *putenv()* don't update *os.environ*, so it is actually preferable to assign to items of *os.environ*. This also applies to *getenv()* and *getenvb()*, which respectively use *os.environ* and *os.environb* in their implementations.

也請見*os.reload_environ()* 函式。

> **備🄵**
>
> On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

引發一個附帶引數 `key`、`value` 的稽核事件 `os.putenv`。

在 3.9 版的變更: 此函式現在都會是可用的。

os.**setegid**(*egid*, /)

Set the current process's effective group id.

可用性: Unix, not WASI, not Android.

os.**seteuid**(*euid*, /)

Set the current process's effective user id.

可用性: Unix, not WASI, not Android.

os.**setgid**(*gid*, /)

Set the current process' group id.

可用性: Unix, not WASI, not Android.

os.**setgroups**(*groups*, /)

Set the list of supplemental group ids associated with the current process to *groups*. *groups* must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

可用性: Unix, not WASI.

備🄵

On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for *getgroups()* for cases where it may not return the same group list set by calling setgroups().

os.**setns**(*fd*, *nstype=0*)

Reassociate the current thread with a Linux namespace. See the *setns(2)* and *namespaces(7)* man pages for more details.

If *fd* refers to a `/proc/pid/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the *CLONE_NEW* constants* to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from *pidfd_open()*. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the *CLONE_NEW* constants*, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller's memberships in unspecified namespaces are left unchanged.

fd can be any object with a *fileno()* method, or a raw file descriptor.

This example reassociates the thread with the `init` process's network namespace:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

可用性: Linux >= 3.0 with glibc >= 2.14.

在 3.12 版被加入.

也參考

unshare() 函式。

os.**setpgrp**()

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

可用性: Unix, not WASI.

os.**setpgid**(*pid*, *pgrp*, /)

Call the system call `setpgid()` to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

可用性: Unix, not WASI.

os.**setpriority**(*which*, *who*, *priority*)

Set program scheduling priority. The value *which* is one of *PRIO_PROCESS*, *PRIO_PGRP*, or *PRIO_USER*, and *who* is interpreted relative to *which* (a process identifier for *PRIO_PROCESS*, process group identifier for *PRIO_PGRP*, and a user ID for *PRIO_USER*). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. *priority* is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.**setregid**(*rgid*, *egid*, /)

Set the current process's real and effective group ids.

可用性: Unix, not WASI, not Android.

os.**setresgid**(*rgid*, *egid*, *sgid*, /)

Set the current process's real, effective, and saved group ids.

可用性: Unix, not WASI, not Android.

在 3.2 版被加入.

os.**setresuid**(*ruid*, *euid*, *suid*, /)

Set the current process's real, effective, and saved user ids.

可用性: Unix, not WASI, not Android.

在 3.2 版被加入.

os.**setreuid**(*ruid*, *euid*, /)

Set the current process's real and effective user ids.

可用性: Unix, not WASI, not Android.

os.**getsid**(*pid*, /)

Call the system call `getsid()`. See the Unix manual for the semantics.

可用性: Unix, not WASI.

os.**setsid**()

Call the system call `setsid()`. See the Unix manual for the semantics.

可用性: Unix, not WASI.

os.**setuid**(*uid*, /)

Set the current process's user id.

可用性: Unix, not WASI, not Android.

os.**strerror**(*code*, /)

Return the error message corresponding to the error code in *code*. On platforms where `strerror()` returns `NULL` when given an unknown error number, *ValueError* is raised.

os.**supports_bytes_environ**

`True` if the native OS type of the environment is bytes (eg. `False` on Windows).

在 3.2 版被加入.

os.**umask**(*mask*, /)

Set the current numeric umask and return the previous umask.

The function is a stub on WASI, see *WebAssembly 平台* for more information.

os.**uname**()

Returns information identifying the current operating system. The return value is an object with five attributes:

- sysname - 作業系統名稱
- nodename - name of machine on network (implementation-defined)
- release - operating system release
- version - 作業系統版本
- machine - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing sysname, nodename, release, version, and machine in that order.

Some systems truncate nodename to 8 characters or to the leading component; a better way to get the hostname is *socket.gethostname()* or even socket.gethostbyaddr(socket.gethostname()).

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., 'Darwin' on macOS and iOS; 'Linux' on Android). *platform.uname()* can be used to get the user-facing operating system name and version on iOS and Android.

可用性: Unix.

在 3.3 版的變更: Return type changed from a tuple to a tuple-like object with named attributes.

os.**unsetenv**(*key*, /)

Unset (delete) the environment variable named *key*. Such changes to the environment affect subprocesses started with *os.system()*, *popen()* or *fork()* and *execv()*.

Deletion of items in *os.environ* is automatically translated into a corresponding call to *unsetenv()*; however, calls to *unsetenv()* don't update *os.environ*, so it is actually preferable to delete items of *os.environ*.

也請見*os.reload_environ()* 函式。

引發一個附帶引數 key 的稽核事件 os.unsetenv。

在 3.9 版的變更: The function is now always available and is also available on Windows.

os.**unshare**(*flags*)

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *unshare(2)* man page for more details. The *flags* argument is a bit mask, combining zero or more of the *CLONE_* constants*, that specifies which parts of the execution context should be unshared from their existing associations and moved to a new namespace. If the *flags* argument is 0, no changes are made to the calling process's execution context.

可用性: Linux >= 2.6.16.

在 3.12 版被加入.

也參考

setns() 函式。

Flags to the *unshare()* function, if the implementation supports them. See *unshare(2)* in the Linux manual for their exact effect and availability.

os.**CLONE_FILES**

os.**CLONE_FS**

os.**CLONE_NEWCGROUP**

os.**CLONE_NEWIPC**

os.**CLONE_NEWNET**

os.**CLONE_NEWNS**

os.**CLONE_NEWPID**

os.**CLONE_NEWTIME**

os.**CLONE_NEWUSER**

os.**CLONE_NEWUTS**

os.**CLONE_SIGHAND**

os.**CLONE_SYSVSEM**

os.**CLONE_THREAD**

os.**CLONE_VM**

16.1.4 File Object Creation

These functions create new *file objects*. (See also `open()` for opening file descriptors.)

os.**fdopen**(*fd*, **args*, ***kwargs*)

Return an open file object connected to the file descriptor *fd*. This is an alias of the `open()` built-in function and accepts the same arguments. The only difference is that the first argument of `fdopen()` must always be an integer.

16.1.5 File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name ”file descriptor” is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a *file object* when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.**close**(*fd*)

Close file descriptor *fd*.

> **備🄵**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a ”file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

os.**closerange**(*fd_low*, *fd_high*, */*)

Close all file descriptors from *fd_low* (inclusive) to *fd_high* (exclusive), ignoring errors. Equivalent to (but much faster than):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an *OSError* is raised with *errno* set to *errno.EXDEV*.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

The return value is the amount of bytes copied. This could be less than the amount requested.

> **備🄵**
>
> On Linux, *os.copy_file_range()* should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

可用性: Linux >= 4.5 with glibc >= 2.27.

在 3.8 版被加入.

os.**device_encoding**(*fd*)

Return a string describing the encoding of the device associated with *fd* if it is connected to a terminal; else return *None*.

On Unix, if the *Python UTF-8 Mode* is enabled, return `'UTF-8'` rather than the device encoding.

在 3.10 版的變更: On Unix, the function now implements the Python UTF-8 Mode.

os.**dup**(*fd*, */*)

Return a duplicate of file descriptor *fd*. The new file descriptor is *non-inheritable*.

On Windows, when duplicating a standard stream (0: stdin, 1: stdout, 2: stderr), the new file descriptor is *inheritable*.

可用性: not WASI.

在 3.4 版的變更: The new file descriptor is now non-inheritable.

os.**dup2**(*fd*, *fd2*, *inheritable=True*)

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Return *fd2*. The new file descriptor is *inheritable* by default or non-inheritable if *inheritable* is `False`.

可用性: not WASI.

在 3.4 版的變更: 新增可選用的參數 *inheritable*。

在 3.7 版的變更: Return *fd2* on success. Previously, `None` was always returned.

os.**fchmod**(*fd*, *mode*)

Change the mode of the file given by *fd* to the numeric *mode*. See the docs for *chmod()* for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

引發一個附帶引數 `path`、`mode`、`dir_fd` 的稽核事件 `os.chmod`。

可用性: Unix, Windows.

The function is limited on WASI, see *WebAssembly 平台* for more information.

在 3.13 版的變更: 新增對 Windows 的支援。

os.**fchown**(*fd*, *uid*, *gid*)

Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1. See *chown()*. As of Python 3.3, this is equivalent to `os.chown(fd, uid, gid)`.

引發一個附帶引數 `path`、`uid`、`gid`、`dir_fd` 的稽核事件 `os.chown`。

可用性: Unix.

The function is limited on WASI, see *WebAssembly 平台* for more information.

os.**fdatasync**(*fd*)

Force write of file with filedescriptor *fd* to disk. Does not force update of metadata.

可用性: Unix.

> **備🄵**
>
> This function is not available on MacOS.

os.**fpathconf**(*fd*, *name*, /)

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, *ValueError* is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an *OSError* is raised with *errno.EINVAL* for the error number.

As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

可用性: Unix.

os.**fstat**(*fd*)

Get the status of the file descriptor *fd*. Return a *stat_result* object.

As of Python 3.3, this is equivalent to `os.stat(fd)`.

> **也參考**
>
> The *stat()* function.

os.**fstatvfs**(*fd*, /)

Return information about the filesystem containing the file associated with file descriptor *fd*, like *statvfs()*. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

可用性: Unix.

os.**fsync**(*fd*)

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `_commit()` function.

If you're starting with a buffered Python *file object* *f*, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with *f* are written to disk.

可用性: Unix, Windows.

os.**ftruncate**(*fd*, *length*, /)

Truncate the file corresponding to file descriptor *fd*, so that it is at most *length* bytes in size. As of Python 3.3, this is equivalent to `os.truncate(fd, length)`.

引發一個附帶引數 `fd`、`length` 的稽核事件 `os.truncate`。

可用性: Unix, Windows.

在 3.5 版的變更: 新增對 Windows 的支援

os.**get_blocking**(*fd*, /)

Get the blocking mode of the file descriptor: `False` if the *O_NONBLOCK* flag is set, `True` if the flag is cleared.

另請參🄵*set_blocking()* 與*socket.socket.setblocking()*。

可用性: Unix, Windows.

The function is limited on WASI, see *WebAssembly 平台* for more information.

On Windows, this function is limited to pipes.

在 3.5 版被加入.

在 3.12 版的變更: 新增對 Windows 上的 pipe 支援。

os.**grantpt**(*fd*, /)

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `grantpt()`.

可用性: Unix, not WASI.

在 3.13 版被加入.

os.**isatty**(*fd*, /)

Return `True` if the file descriptor *fd* is open and connected to a tty(-like) device, else `False`.

os.**lockf**(*fd*, *cmd*, *len*, /)

Apply, test or remove a POSIX lock on an open file descriptor. *fd* is an open file descriptor. *cmd* specifies the command to use - one of *F_LOCK*, *F_TLOCK*, *F_ULOCK* or *F_TEST*. *len* specifies the section of the file to lock.

引發一個附帶引數 `fd`、`cmd`、`len` 的稽核事件 `os.lockf`。

可用性: Unix.

在 3.3 版被加入.

os.**F_LOCK**

os.**F_TLOCK**

os.**F_ULOCK**

os.**F_TEST**

Flags that specify what action *lockf()* will take.

可用性: Unix.

在 3.3 版被加入.

os.**login_tty**(*fd*, /)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process; close fd.

可用性: Unix, not WASI.

在 3.11 版被加入.

os.**lseek**(*fd*, *pos*, *whence*, /)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are:

- *SEEK_SET* or `0` -- set *pos* relative to the beginning of the file
- *SEEK_CUR* or `1` -- set *pos* relative to the current file position

- *SEEK_END* or 2 -- set *pos* relative to the end of the file
- *SEEK_HOLE* -- set *pos* to the next data location, relative to *pos*
- *SEEK_DATA* -- set *pos* to the next data hole, relative to *pos*

在 3.3 版的變更: Add support for SEEK_HOLE and SEEK_DATA.

os.**SEEK_SET**

os.**SEEK_CUR**

os.**SEEK_END**

Parameters to the *lseek()* function and the *seek()* method on *file-like objects*, for whence to adjust the file position indicator.

SEEK_SET
Adjust the file position relative to the beginning of the file.

SEEK_CUR
Adjust the file position relative to the current file position.

SEEK_END
Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.**SEEK_HOLE**

os.**SEEK_DATA**

Parameters to the *lseek()* function and the *seek()* method on *file-like objects*, for seeking file data and holes on sparsely allocated files.

SEEK_DATA
Adjust the file offset to the next location containing data, relative to the seek position.

SEEK_HOLE
Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

> **備🄵**
>
> These operations only make sense for filesystems that support them.

可用性: Linux >= 3.1, macOS, Unix

在 3.3 版被加入.

os.**open**(*path*, *flags*, *mode=0o777*, ***, *dir_fd=None*)

Open the file *path* and set various flags according to *flags* and possibly its mode according to *mode*. When computing *mode*, the current umask value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is *non-inheritable*.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like *O_RDONLY* and *O_WRONLY*) are defined in the *os* module. In particular, on Windows adding *O_BINARY* is needed to open files in binary mode.

This function can support *paths relative to directory descriptors* with the *dir_fd* parameter.

引發一個附帶引數 path、mode、flags 的稽核事件 open。

在 3.4 版的變更: The new file descriptor is now non-inheritable.

> **備🄵**
>
> This function is intended for low-level I/O. For normal usage, use the built-in function `open()`, which returns a *file object* with `read()` and `write()` methods (and many more). To wrap a file descriptor in a file object, use `fdopen()`.

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

在 3.6 版的變更: Accepts a *path-like object*.

The following constants are options for the *flags* parameter to the `open()` function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the `open(2)` manual page on Unix or the MSDN on Windows.

os.O_RDONLY

os.O_WRONLY

os.O_RDWR

os.O_APPEND

os.O_CREAT

os.O_EXCL

os.O_TRUNC

The above constants are available on Unix and Windows.

os.O_DSYNC

os.O_RSYNC

os.O_SYNC

os.O_NDELAY

os.O_NONBLOCK

os.O_NOCTTY

os.O_CLOEXEC

The above constants are only available on Unix.

在 3.3 版的變更: Add `O_CLOEXEC` constant.

os.O_BINARY

os.O_NOINHERIT

os.O_SHORT_LIVED

os.O_TEMPORARY

os.O_RANDOM

os.O_SEQUENTIAL

os.O_TEXT

The above constants are only available on Windows.

os.O_EVTONLY

os.O_FSYNC

os.O_SYMLINK

os.O_NOFOLLOW_ANY

The above constants are only available on macOS.

在 3.10 版的變更: Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` constants.

os.O_ASYNC

os.**O_DIRECT**

os.**O_DIRECTORY**

os.**O_NOFOLLOW**

os.**O_NOATIME**

os.**O_PATH**

os.**O_TMPFILE**

os.**O_SHLOCK**

os.**O_EXLOCK**

The above constants are extensions and not present if they are not defined by the C library.

在 3.4 版的變更: Add *O_PATH* on systems that support it. Add *O_TMPFILE*, only available on Linux Kernel 3.11 or newer.

os.**openpty**()

Open a new pseudo-terminal pair. Return a pair of file descriptors `(master, slave)` for the pty and the tty, respectively. The new file descriptors are *non-inheritable*. For a (slightly) more portable approach, use the *pty* module.

可用性: Unix, not WASI.

在 3.4 版的變更: The new file descriptors are now non-inheritable.

os.**pipe**()

Create a pipe. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively. The new file descriptor is *non-inheritable*.

可用性: Unix, Windows.

在 3.4 版的變更: The new file descriptors are now non-inheritable.

os.**pipe2**(*flags*, */*)

Create a pipe with *flags* set atomically. *flags* can be constructed by ORing together one or more of these values: *O_NONBLOCK*, *O_CLOEXEC*. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.**posix_fallocate**(*fd*, *offset*, *len*, */*)

Ensures that enough disk space is allocated for the file specified by *fd* starting from *offset* and continuing for *len* bytes.

可用性: Unix.

在 3.3 版被加入.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, */*)

Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by *fd* starting at *offset* and continuing for *len* bytes. *advice* is one of *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* or *POSIX_FADV_DONTNEED*.

可用性: Unix.

在 3.3 版被加入.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

Flags that can be used in *advice* in *posix_fadvise()* that specify the access pattern that is likely to be used.

可用性: Unix.

在 3.3 版被加入.

os.**pread**(*fd*, *n*, *offset*, /)

Read at most *n* bytes from file descriptor *fd* at a position of *offset*, leaving the file offset unchanged.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

可用性: Unix.

在 3.3 版被加入.

os.**posix_openpt**(*oflag*, /)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function `posix_openpt()`. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of `posix_openpt()` of your system.

The returned file descriptor is *non-inheritable*. If the value *O_CLOEXEC* is available on the system, it is added to *oflag*.

可用性: Unix, not WASI.

在 3.13 版被加入.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*, /)

Read from a file descriptor *fd* at a position of *offset* into mutable *bytes-like objects* *buffers*, leaving the file offset unchanged. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

The flags argument contains a bitwise OR of zero or more of the following flags:

- *RWF_HIPRI*
- *RWF_NOWAIT*

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of *os.readv()* and *os.pread()*.

可用性: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

在 3.7 版被加入.

os.**RWF_NOWAIT**

Do not wait for data which is not immediately available. If this flag is specified, the system call will return instantly if it would have to read data from the backing storage or wait for a lock.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to *errno.EAGAIN*.

可用性: Linux >= 4.14.

在 3.7 版被加入.

os.**RWF_HIPRI**

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources.

Currently, on Linux, this feature is usable only on a file descriptor opened using the *O_DIRECT* flag.

可用性: Linux >= 4.6.

在 3.7 版被加入.

os.**ptsname**(*fd*, */*)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function `ptsname_r()` if it is available; otherwise, the C standard library function `ptsname()`, which is not guaranteed to be thread-safe, is called.

可用性: Unix, not WASI.

在 3.13 版被加入.

os.**pwrite**(*fd*, *str*, *offset*, */*)

Write the bytestring in *str* to file descriptor *fd* at position of *offset*, leaving the file offset unchanged.

Return the number of bytes actually written.

可用性: Unix.

在 3.3 版被加入.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*, */*)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

The flags argument contains a bitwise OR of zero or more of the following flags:

- *RWF_DSYNC*
- *RWF_SYNC*
- *RWF_APPEND*

Return the total number of bytes actually written.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of *os.writev()* and *os.pwrite()*.

可用性: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

在 3.7 版被加入.

os.**RWF_DSYNC**

Provide a per-write equivalent of the *O_DSYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

可用性: Linux >= 4.7.

在 3.7 版被加入.

os.**RWF_SYNC**

Provide a per-write equivalent of the *O_SYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

可用性: Linux >= 4.7.

在 3.7 版被加入.

os.**RWF_APPEND**

Provide a per-write equivalent of the *O_APPEND* *os.open()* flag. This flag is meaningful only for *os.pwritev()*, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

可用性: Linux >= 4.16.

在 3.10 版被加入.

os.**read**(*fd*, *n*, */*)

Read at most *n* bytes from file descriptor *fd*.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

> **備🄵**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by *os.open()* or *pipe()*. To read a "file object" returned by the built-in function *open()* or by *popen()* or *fdopen()*, or *sys.stdin*, use its `read()` or `readline()` methods.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

os.**readinto**(*fd*, *buffer*, */*)

Read from a file descriptor *fd* into a mutable buffer object *buffer*.

The *buffer* should be mutable and *bytes-like*. On success, returns the number of bytes read. Less bytes may be read than the size of the buffer. The underlying system call will be retried when interrupted by a signal, unless the signal handler raises an exception. Other errors will not be retried and an error will be raised.

Returns 0 if *fd* is at end of file or if the provided *buffer* has length 0 (which can be used to check for errors without reading data). Never returns negative.

> **備🄵**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by *os.open()* or *os.pipe()*. To read a "file object" returned by the built-in function *open()*, or *sys.stdin*, use its member functions, for example *io.BufferedIOBase.readinto()*, *io.BufferedIOBase.read()*, or *io.TextIOBase.read()*

在 3.14 版被加入.

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*)

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

Copy *count* bytes from file descriptor *in_fd* to file descriptor *out_fd* starting at *offset*. Return the number of bytes sent. When EOF is reached return `0`.

The first function notation is supported by all platforms that define *sendfile()*.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in_fd* and the position of *in_fd* is updated.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

All platforms support sockets as *out_fd* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Cross-platform applications should not use *headers*, *trailers* and *flags* arguments.

可用性: Unix, not WASI.

> **備🄵**
>
> For a higher-level wrapper of `sendfile()`, see `socket.socket.sendfile()`.

在 3.3 版被加入.

在 3.9 版的變更: Parameters *out* and *in* was renamed to *out_fd* and *in_fd*.

os.SF_NODISKIO

os.SF_MNOWAIT

os.SF_SYNC

Parameters to the `sendfile()` function, if the implementation supports them.

可用性: Unix, not WASI.

在 3.3 版被加入.

os.SF_NOCACHE

Parameter to the `sendfile()` function, if the implementation supports it. The data won't be cached in the virtual memory and will be freed afterwards.

可用性: Unix, not WASI.

在 3.11 版被加入.

os.set_blocking(*fd*, *blocking*, */*)

Set the blocking mode of the specified file descriptor. Set the `O_NONBLOCK` flag if blocking is `False`, clear the flag otherwise.

另請參🄵`get_blocking()` 與`socket.socket.setblocking()`。

可用性: Unix, Windows.

The function is limited on WASI, see *WebAssembly 平台* for more information.

On Windows, this function is limited to pipes.

在 3.5 版被加入.

在 3.12 版的變更: 新增對 Windows 上的 pipe 支援。

os.splice(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*, *flags=0*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*.

The splicing behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- If `SPLICE_F_MOVE` is specified, the kernel is asked to move pages instead of copying, but pages may still be copied if the kernel cannot move the pages from the pipe.
- If `SPLICE_F_NONBLOCK` is specified, the kernel is asked to not block on I/O. This makes the splice pipe operations nonblocking, but splice may nevertheless block because the spliced file descriptors may block.
- If `SPLICE_F_MORE` is specified, it hints to the kernel that more data will be coming in a subsequent splice.

At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations. The copy is done as if both files are opened as binary.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

也參考

splice(2) 使用手🄵。

可用性: Linux >= 2.6.17 with glibc >= 2.5

在 3.10 版被加入.

os.**SPLICE_F_MOVE**

os.**SPLICE_F_NONBLOCK**

os.**SPLICE_F_MORE**

在 3.10 版被加入.

os.**readv**(*fd*, *buffers*, /)

Read from a file descriptor *fd* into a number of mutable *bytes-like objects* *buffers*. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

可用性: Unix.

在 3.3 版被加入.

os.**tcgetpgrp**(*fd*, /)

Return the process group associated with the terminal given by *fd* (an open file descriptor as returned by *os.open()*).

可用性: Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, /)

Set the process group associated with the terminal given by *fd* (an open file descriptor as returned by *os.open()*) to *pg*.

可用性: Unix, not WASI.

os.**ttyname**(*fd*, /)

Return a string which specifies the terminal device associated with file descriptor *fd*. If *fd* is not associated with a terminal device, an exception is raised.

可用性: Unix.

os.**unlockpt**(*fd*, /)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `unlockpt()`.

可用性: Unix, not WASI.

在 3.13 版被加入.

os.**write**(*fd*, *str*, /)

Write the bytestring in *str* to file descriptor *fd*.

Return the number of bytes actually written.

> **備🄵**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by *os.open()* or *pipe()*. To write a "file object" returned by the built-in function *open()* or by *popen()* or *fdopen()*, or *sys.stdout* or *sys.stderr*, use its `write()` method.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

os.**writev**(*fd*, *buffers*, /)

Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

Returns the total number of bytes actually written.

The operating system may set a limit (*sysconf()* value `'SC_IOV_MAX'`) on the number of buffers that can be used.

可用性: Unix.

在 3.3 版被加入.

Querying the size of a terminal

在 3.3 版被加入.

os.**get_terminal_size**(*fd=STDOUT_FILENO*, /)

Return the size of the terminal window as `(columns, lines)`, tuple of type *terminal_size*.

The optional argument `fd` (default `STDOUT_FILENO`, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an *OSError* is raised.

shutil.get_terminal_size() is the high-level function which should normally be used, `os.get_terminal_size` is the low-level implementation.

可用性: Unix, Windows.

class os.**terminal_size**

A subclass of tuple, holding `(columns, lines)` of the terminal window size.

columns

Width of the terminal window in characters.

lines

Height of the terminal window in characters.

Inheritance of File Descriptors

在 3.4 版被加入.

A file descriptor has an "inheritable" flag which indicates if the file descriptor can be inherited by child processes. Since Python 3.4, file descriptors created by Python are non-inheritable by default.

On UNIX, non-inheritable file descriptors are closed in child processes at the execution of a new program, other file descriptors are inherited.

On Windows, non-inheritable handles and file descriptors are closed in child processes, except for standard streams (file descriptors 0, 1 and 2: stdin, stdout and stderr), which are always inherited. Using *spawn** functions, all inheritable handles and all inheritable file descriptors are inherited. Using the *subprocess* module, all file descriptors except standard streams are closed, and inheritable handles are only inherited if the *close_fds* parameter is `False`.

On WebAssembly platforms, the file descriptor cannot be modified.

`os.`**`get_inheritable`**(*fd*, */*)

Get the "inheritable" flag of the specified file descriptor (a boolean).

`os.`**`set_inheritable`**(*fd*, *inheritable*, */*)

Set the "inheritable" flag of the specified file descriptor.

`os.`**`get_handle_inheritable`**(*handle*, */*)

Get the "inheritable" flag of the specified handle (a boolean).

可用性: Windows.

`os.`**`set_handle_inheritable`**(*handle*, *inheritable*, */*)

Set the "inheritable" flag of the specified handle.

可用性: Windows.

16.1.6 Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** Normally the *path* argument provided to functions in the *os* module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).

  You can check whether or not *path* can be specified as a file descriptor for a particular function on your platform using *os.supports_fd*. If this functionality is unavailable, using it will raise a *NotImplementedError*.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it's an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

  You can check whether or not *dir_fd* is supported for a particular function on your platform using *os.supports_dir_fd*. If it's unavailable, using it will raise a *NotImplementedError*.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. For POSIX systems, Python will call the `l...` variant of the function.

  You can check whether or not *follow_symlinks* is supported for a particular function on your platform using *os.supports_follow_symlinks*. If it's unavailable, using it will raise a *NotImplementedError*.

`os.`**`access`**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Use the real uid/gid to test for access to *path*. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to *path*. *mode* should be *F_OK* to test the existence of *path*, or it can be the inclusive OR of one or more of *R_OK*, *W_OK*, and *X_OK* to test permissions. Return *True* if access is allowed, *False* if not. See the Unix man page *access(2)* for more information.

This function can support specifying *paths relative to directory descriptors* and *not following symlinks*.

If *effective_ids* is `True`, *access()* will perform its access checks using the effective uid/gid instead of the real uid/gid. *effective_ids* may not be supported on your platform; you can check whether or not it is available using *os.supports_effective_ids*. If it is unavailable, using it will raise a *NotImplementedError*.

備🄵

Using *access()* to check if a user is authorized to e.g. open a file before actually doing so using *open()* creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use *EAFP* techniques. For example:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

is better written as:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

備🄵

I/O operations may fail even when *access()* indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

在 3.3 版的變更: 新增 *dir_fd*、*effective_ids* 與 *follow_symlinks* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

os.**F_OK**

os.**R_OK**

os.**W_OK**

os.**X_OK**

Values to pass as the *mode* parameter of *access()* to test the existence, readability, writability and executability of *path*, respectively.

os.**chdir**(*path*)

Change the current working directory to *path*.

This function can support *specifying a file descriptor*. The descriptor must refer to an opened directory, not an open file.

This function can raise *OSError* and subclasses such as *FileNotFoundError*, *PermissionError*, and *NotADirectoryError*.

引發一個附帶引數 `path` 的稽核事件 `os.chdir`。

在 3.3 版的變更: Added support for specifying *path* as a file descriptor on some platforms.

在 3.6 版的變更: Accepts a *path-like object*.

os.**chflags**(*path*, *flags*, *, *follow_symlinks=True*)

Set the flags of *path* to the numeric *flags*. *flags* may take a combination (bitwise OR) of the following values (as defined in the *stat* module):

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*

- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*
- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

This function can support *not following symlinks*.

引發一個附帶引數 `path`、`flags` 的稽核事件 `os.chflags`。

可用性: Unix, not WASI.

在 3.3 版的變更: 新增 *follow_symlinks* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

`os.`**`chmod`**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the *stat* module) or bitwise ORed combinations of them:

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

> **備🄵**
>
> Although Windows supports *chmod()*, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.
>
> The function is limited on WASI, see *WebAssembly 平台* for more information.

引發一個附帶引數 `path`、`mode`、`dir_fd` 的稽核事件 `os.chmod`。

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.13 版的變更: Added support for a file descriptor and the *follow_symlinks* argument on Windows.

`os.`**`chown`**(*path*, *uid*, *gid*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

See *shutil.chown()* for a higher-level function that accepts names in addition to numeric ids.

引發一個附帶引數 `path`、`uid`、`gid`、`dir_fd` 的稽核事件 `os.chown`。

可用性: Unix.

The function is limited on WASI, see *WebAssembly 平台* for more information.

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

在 3.6 版的變更: Supports a *path-like object*.

`os.`**`chroot`**(*path*)

Change the root directory of the current process to *path*.

可用性: Unix, not WASI, not Android.

在 3.6 版的變更: Accepts a *path-like object*.

`os.`**`fchdir`**(*fd*)

Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file. As of Python 3.3, this is equivalent to `os.chdir(fd)`.

引發一個附帶引數 `path` 的稽核事件 `os.chdir`。

可用性: Unix.

`os.`**`getcwd`**()

Return a string representing the current working directory.

`os.`**`getcwdb`**()

Return a bytestring representing the current working directory.

在 3.8 版的變更: The function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** for the rationale. The function is no longer deprecated on Windows.

`os.`**`lchflags`**(*path*, *flags*)

Set the flags of *path* to the numeric *flags*, like *chflags()*, but do not follow symbolic links. As of Python 3.3, this is equivalent to `os.chflags(path, flags, follow_symlinks=False)`.

引發一個附帶引數 `path`、`flags` 的稽核事件 `os.chflags`。

可用性: Unix, not WASI.

在 3.6 版的變更: Accepts a *path-like object*.

os.**lchmod**(*path*, *mode*)

Change the mode of *path* to the numeric *mode*. If path is a symlink, this affects the symlink rather than the target. See the docs for *chmod()* for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(path, mode, follow_symlinks=False)`.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

引發一個附帶引數 `path`、`mode`、`dir_fd` 的稽核事件 `os.chmod`。

可用性: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

在 3.6 版的變更: Accepts a *path-like object*.

在 3.13 版的變更: 新增對 Windows 的支援。

os.**lchown**(*path*, *uid*, *gid*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. This function will not follow symbolic links. As of Python 3.3, this is equivalent to `os.chown(path, uid, gid, follow_symlinks=False)`.

引發一個附帶引數 `path`、`uid`、`gid`、`dir_fd` 的稽核事件 `os.chown`。

可用性: Unix.

在 3.6 版的變更: Accepts a *path-like object*.

os.**link**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

Create a hard link pointing to *src* named *dst*.

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*, and *not following symlinks*. The default value of *follow_symlinks* is `False` on Windows.

引發一個附帶引數 `src`、`dst`、`src_dir_fd`、`dst_dir_fd` 的稽核事件 `os.link`。

可用性: Unix, Windows.

在 3.2 版的變更: 新支援 Windows。

在 3.3 版的變更: 新增 *src_dir_fd*、*dst_dir_fd* 與 *follow_symlinks* 參數。

在 3.6 版的變更: Accepts a *path-like object* for *src* and *dst*.

os.**listdir**(*path='.'*)

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path may be a *path-like object*. If *path* is of type `bytes` (directly or indirectly through the *PathLike* interface), the filenames returned will also be of type `bytes`; in all other circumstances, they will be of type `str`.

This function can also support *specifying a file descriptor*; the file descriptor must refer to a directory.

引發一個附帶引數 `path` 的稽核事件 `os.listdir`。

備𝔽

To encode `str` filenames to `bytes`, use *fsencode()*.

也參考

The *scandir()* function returns directory entries along with file attribute information, giving better performance for many common use cases.

在 3.2 版的變更: The *path* parameter became optional.

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor.

在 3.6 版的變更: Accepts a *path-like object*.

os.**listdrives**()

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise *OSError* if an error occurs collecting the drive names.

引發一個不附帶引數的稽核事件 `os.listdrives`。

可用性: Windows

在 3.12 版被加入.

os.**listmounts**(*volume*)

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by *os.listvolumes()*. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises *OSError* if the volume is not recognized or if an error occurs collecting the paths.

引發一個附帶引數 `volume` 的稽核事件 `os.listmounts`。

可用性: Windows

在 3.12 版被加入.

os.**listvolumes**()

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using *os.listmounts()*.

May raise *OSError* if an error occurs collecting the volumes.

引發一個不附帶引數的稽核事件 `os.listvolumes`。

可用性: Windows

在 3.12 版被加入.

os.**lstat**(*path*, *, *dir_fd=None*)

Perform the equivalent of an `lstat()` system call on the given path. Similar to *stat()*, but does not follow symbolic links. Return a *stat_result* object.

On platforms that do not support symbolic links, this is an alias for *stat()*.

As of Python 3.3, this is equivalent to `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

This function can also support *paths relative to directory descriptors*.

> **也參考**
>
> The *stat()* function.

在 3.2 版的變更: Added support for Windows 6.0 (Vista) symbolic links.

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

在 3.8 版的變更: On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for `stat()`.

`os.`**`mkdir`**(*path*, *mode=0o777*, ***, *dir_fd=None*)

Create a directory named *path* with numeric mode *mode*.

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the *mode*) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call `chmod()` explicitly to set them.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

This function can also support *paths relative to directory descriptors*.

It is also possible to create temporary directories; see the `tempfile` module's `tempfile.mkdtemp()` function.

引發一個附帶引數 `path`、`mode`、`dir_fd` 的稽核事件 `os.mkdir`。

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

在 3.13 版的變更: Windows now handles a *mode* of `0o700`.

`os.`**`makedirs`**(*name*, *mode=0o777*, *exist_ok=False*)

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The *mode* parameter is passed to `mkdir()` for creating the leaf directory; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a `FileExistsError` is raised if the target directory already exists.

> **備🄵**
>
> `makedirs()` will become confused if the path elements to create include `pardir` (eg. ".." on UNIX systems).

This function handles UNC paths correctly.

引發一個附帶引數 `path`、`mode`、`dir_fd` 的稽核事件 `os.mkdir`。

在 3.2 版的變更: 新增 *exist_ok* 參數。

在 3.4.1 版的變更: Before Python 3.4.1, if *exist_ok* was `True` and the directory existed, `makedirs()` would still raise an error if *mode* did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See bpo-21082.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.7 版的變更: The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

os.**mkfifo**(*path*, *mode=0o666*, *, *dir_fd=None*)

Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The current umask value is first masked out from the mode.

This function can also support *paths relative to directory descriptors*.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn't open the FIFO --- it just creates the rendezvous point.

可用性: Unix, not WASI.

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

os.**mknod**(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Create a filesystem node (file, device special file or named pipe) named *path*. *mode* specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in *stat*). For `stat.S_IFCHR` and `stat.S_IFBLK`, *device* defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

This function can also support *paths relative to directory descriptors*.

可用性: Unix, not WASI.

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

os.**major**(*device*, /)

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**minor**(*device*, /)

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**makedev**(*major*, *minor*, /)

Compose a raw device number from the major and minor device numbers.

os.**pathconf**(*path*, *name*)

Return system configuration information relevant to a named file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

This function can support *specifying a file descriptor*.

可用性: Unix.

在 3.6 版的變更: Accepts a *path-like object*.

os.**pathconf_names**

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

可用性: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the *path* is a string object (directly or indirectly through a *PathLike* interface), the result will also be a string object, and the call may raise a UnicodeDecodeError. If the *path* is a bytes object (direct or indirectly), the result will be a bytes object.

This function can also support *paths relative to directory descriptors*.

When trying to resolve a path that may contain links, use *realpath()* to properly handle recursion and platform differences.

可用性: Unix, Windows.

在 3.2 版的變更: Added support for Windows 6.0 (Vista) symbolic links.

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object* on Unix.

在 3.8 版的變更: Accepts a *path-like object* and a bytes object on Windows.

Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional "print name" field that was previously returned.

os.**remove**(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an *OSError* is raised. Use *rmdir()* to remove directories. If the file does not exist, a *FileNotFoundError* is raised.

This function can support *paths relative to directory descriptors*.

On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is semantically identical to *unlink()*.

引發一個附帶引數 `path`、`dir_fd` 的稽核事件 `os.remove`。

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

os.**removedirs**(*name*)

Remove directories recursively. Works like *rmdir()* except that, if the leaf directory is successfully removed, *removedirs()* tries to successively remove every parent directory mentioned in *path* until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory `'foo/bar/baz'`, and then remove `'foo/bar'` and `'foo'` if they are empty. Raises *OSError* if the leaf directory could not be successfully removed.

引發一個附帶引數 `path`、`dir_fd` 的稽核事件 `os.remove`。

在 3.6 版的變更: Accepts a *path-like object*.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an *OSError* subclass in a number of cases:

On Windows, if *dst* exists a *FileExistsError* is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use *shutil.move()* to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an *IsADirectoryError* or a *NotADirectoryError* will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an *OSError* is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*.

If you want cross-platform overwriting of the destination, use `replace()`.

引發一個附帶引數 `src`、`dst`、`src_dir_fd`、`dst_dir_fd` 的稽核事件 `os.rename`。

在 3.3 版的變更: 新增 *src_dir_fd* 與 *dst_dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object* for *src* and *dst*.

`os.`**`renames`**(*old*, *new*)

Recursive directory or file renaming function. Works like `rename()`, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using `removedirs()`.

> **備🄵**
>
> This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

引發一個附帶引數 `src`、`dst`、`src_dir_fd`、`dst_dir_fd` 的稽核事件 `os.rename`。

在 3.6 版的變更: Accepts a *path-like object* for *old* and *new*.

`os.`**`replace`**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*.

引發一個附帶引數 `src`、`dst`、`src_dir_fd`、`dst_dir_fd` 的稽核事件 `os.rename`。

在 3.3 版被加入.

在 3.6 版的變更: Accepts a *path-like object* for *src* and *dst*.

`os.`**`rmdir`**(*path*, *, *dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a `FileNotFoundError` or an `OSError` is raised respectively. In order to remove whole directory trees, `shutil.rmtree()` can be used.

This function can support *paths relative to directory descriptors*.

引發一個附帶引數 `path`、`dir_fd` 的稽核事件 `os.rmdir`。

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

`os.`**`scandir`**(*path='.'*)

Return an iterator of `os.DirEntry` objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

Using `scandir()` instead of `listdir()` can significantly increase the performance of code that also needs file type or file attribute information, because `os.DirEntry` objects expose this information if the operating system provides it when scanning a directory. All `os.DirEntry` methods may perform a system call, but `is_dir()` and `is_file()` usually only require a system call for symbolic links; `os.DirEntry.stat()` always requires a system call on Unix but only requires one for symbolic links on Windows.

path may be a *path-like object*. If *path* is of type `bytes` (directly or indirectly through the `PathLike` interface), the type of the `name` and `path` attributes of each `os.DirEntry` will be `bytes`; in all other circumstances, they will be of type `str`.

This function can also support *specifying a file descriptor*; the file descriptor must refer to a directory.

引發一個附帶引數 `path` 的稽核事件 `os.scandir`。

`scandir()` 🄵代器支援*context manager* 協定，🄵具有以下方法：

`scandir.close()`

Close the iterator and free acquired resources.

This is called automatically when the iterator is exhausted or garbage collected, or when an error happens during iterating. However it is advisable to call it explicitly or use the `with` statement.

在 3.6 版被加入.

The following example shows a simple use of `scandir()` to display all the files (excluding directories) in the given *path* that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

備🄵

On Unix-based systems, `scandir()` uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

在 3.5 版被加入.

在 3.6 版的變更: Added support for the *context manager* protocol and the `close()` method. If a `scandir()` iterator is neither exhausted nor explicitly closed a `ResourceWarning` will be emitted in its destructor.

The function accepts a *path-like object*.

在 3.7 版的變更: 新增對 Unix 上的檔案描述器的支援。

class `os.`**DirEntry**

Object yielded by `scandir()` to expose the file path and other file attributes of a directory entry.

`scandir()` will provide as much of this information as possible without making additional system calls. When a `stat()` or `lstat()` system call is made, the `os.DirEntry` object will cache the result.

`os.DirEntry` instances are not intended to be stored in long-lived data structures; if you know the file metadata has changed or if a long time has elapsed since calling `scandir()`, call `os.stat(entry.path)` to fetch up-to-date information.

Because the `os.DirEntry` methods can make operating system calls, they may also raise `OSError`. If you need very fine-grained control over errors, you can catch `OSError` when calling one of the `os.DirEntry` methods and handle as appropriate.

To be directly usable as a *path-like object*, `os.DirEntry` implements the `PathLike` interface.

Attributes and methods on a `os.DirEntry` instance are as follows:

name

The entry's base filename, relative to the `scandir()` *path* argument.

The `name` attribute will be `bytes` if the `scandir()` *path* argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

path

The entry's full path name: equivalent to `os.path.join(scandir_path, entry.name)` where *scandir_path* is the `scandir()` *path* argument. The path is only absolute if the `scandir()` *path* argument was absolute. If the `scandir()` *path* argument was a *file descriptor*, the `path` attribute is the same as the `name` attribute.

The *path* attribute will be `bytes` if the *scandir()* *path* argument is of type `bytes` and `str` otherwise. Use *fsdecode()* to decode byte filenames.

inode()

Return the inode number of the entry.

The result is cached on the `os.DirEntry` object. Use `os.stat(entry.path, follow_symlinks=False).st_ino` to fetch up-to-date information.

On the first, uncached call, a system call is required on Windows but not on Unix.

is_dir(*, *follow_symlinks=True*)

Return `True` if this entry is a directory or a symbolic link pointing to a directory; return `False` if the entry is or points to any other kind of file, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a directory (without following symlinks); return `False` if the entry is any other kind of file or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call *os.stat()* along with *stat.S_ISDIR()* to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, for non-symlinks, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`. If the entry is a symlink, a system call will be required to follow the symlink unless *follow_symlinks* is `False`.

This method can raise *OSError*, such as *PermissionError*, but *FileNotFoundError* is caught and not raised.

is_file(*, *follow_symlinks=True*)

Return `True` if this entry is a file or a symbolic link pointing to a file; return `False` if the entry is or points to a directory or other non-file entry, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a file (without following symlinks); return `False` if the entry is a directory or other non-file entry, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Caching, system calls made, and exceptions raised are as per *is_dir()*.

is_symlink()

Return `True` if this entry is a symbolic link (even if broken); return `False` if the entry points to a directory or any kind of file, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.islink()* to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`.

This method can raise *OSError*, such as *PermissionError*, but *FileNotFoundError* is caught and not raised.

is_junction()

Return `True` if this entry is a junction (even if broken); return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.isjunction()* to fetch up-to-date information.

在 3.12 版被加入.

stat(*, *follow_symlinks=True*)

Return a *stat_result* object for this entry. This method follows symbolic links by default; to stat a symbolic link add the `follow_symlinks=False` argument.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a reparse point (for example, a symbolic link or directory junction).

On Windows, the `st_ino`, `st_dev` and `st_nlink` attributes of the *stat_result* are always set to zero. Call *os.stat()* to get these attributes.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call *os.stat()* to fetch up-to-date information.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of *pathlib.Path*. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

在 3.5 版被加入.

在 3.6 版的變更: Added support for the *PathLike* interface. Added support for *bytes* paths on Windows.

在 3.12 版的變更: The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

`os.`**`stat`**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

Get the status of a file or a file descriptor. Perform the equivalent of a `stat()` system call on the given path. *path* may be specified as either a string or bytes -- directly or indirectly through the *PathLike* interface -- or as an open file descriptor. Return a *stat_result* object.

This function normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use *lstat()*.

This function can support *specifying a file descriptor* and *not following symlinks*.

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the *os.path.realpath()* function to resolve the path name as far as possible and call *lstat()* on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

範例:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

也參考

fstat() 和*lstat()* 函式。

在 3.3 版的變更: 新增 *dir_fd* 與 *follow_symlinks* 參數，指定一個檔案描述器而非路徑。

在 3.6 版的變更: Accepts a *path-like object*.

在 3.8 版的變更: On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points. If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class os.**stat_result**

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of *os.stat()*, *os.fstat()* and *os.lstat()*.

屬性:

st_mode

File mode: file type and file mode bits (permissions).

st_ino

Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index on Windows

st_dev

Identifier of the device on which this file resides.

st_nlink

Number of hard links.

st_uid

User identifier of the file owner.

st_gid

Group identifier of the file owner.

st_size

Size of the file in bytes, if it is a regular file or a symbolic link. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Timestamps:

st_atime

Time of most recent access expressed in seconds.

st_mtime

Time of most recent content modification expressed in seconds.

st_ctime

Time of most recent metadata change expressed in seconds.

在 3.12 版的變更: `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_atime_ns

Time of most recent access expressed in nanoseconds as an integer.

在 3.3 版被加入.

st_mtime_ns

Time of most recent content modification expressed in nanoseconds as an integer.

在 3.3 版被加入.

st_ctime_ns

Time of most recent metadata change expressed in nanoseconds as an integer.

在 3.3 版被加入.

在 3.12 版的變更: `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise *AttributeError*.

在 3.12 版的變更: `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise *AttributeError*.

在 3.12 版被加入.

備🄵

The exact meaning and resolution of the *st_atime*, *st_mtime*, *st_ctime* and *st_birthtime* attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, *st_mtime* has 2-second resolution, and *st_atime* has only 1-day resolution. See your operating system documentation for details.

Similarly, although *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns* are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store *st_atime*, *st_mtime*, *st_ctime* and *st_birthtime* cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns*.

On some Unix systems (such as Linux), the following attributes may also be available:

st_blocks

Number of 512-byte blocks allocated for file. This may be smaller than *st_size*/512 when the file has holes.

st_blksize

”Preferred” blocksize for efficient file system I/O. Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.

st_rdev

Type of device if an inode device.

st_flags

User defined flags for file.

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

st_gen

File generation number.

On Solaris and derivatives, the following attributes may also be available:

st_fstype

String that uniquely identifies the type of the filesystem that contains the file.

On macOS systems, the following attributes may also be available:

st_rsize

Real size of the file.

st_creator

Creator of the file.

st_type

File type.

On Windows systems, the following attributes are also available:

st_file_attributes

Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>` constants in the *stat* module.

在 3.5 版被加入.

st_reparse_tag

When *st_file_attributes* has the *FILE_ATTRIBUTE_REPARSE_POINT* set, this field contains the tag identifying the type of reparse point. See the *IO_REPARSE_TAG_** constants in the *stat* module.

The standard module *stat* defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a *stat_result* instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

在 3.5 版的變更: Windows now returns the file index as *st_ino* when available.

在 3.7 版的變更: Added the *st_fstype* member to Solaris/derivatives.

在 3.8 版的變更: 在 Windows 上新增*st_reparse_tag* 成員。

在 3.8 版的變更: On Windows, the *st_mode* member now identifies special files as `S_IFCHR`, `S_IFIFO` or `S_IFBLK` as appropriate.

在 3.12 版的變更: On Windows, *st_ctime* is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use *st_birthtime* for the creation time.

On Windows, *st_ino* may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.

On Windows, *st_rdev* no longer returns a value. Previously it would contain the same as *st_dev*, which was incorrect.

在 Windows 上新增*st_birthtime* 成員。

`os.`**statvfs**(*path*)

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Two module-level constants are defined for the `f_flag` attribute's bit-flags: if `ST_RDONLY` is set, the filesystem is mounted read-only, and if `ST_NOSUID` is set, the semantics of setuid/setgid bits are disabled or not supported.

Additional module-level constants are defined for GNU/glibc based systems. These are `ST_NODEV` (disallow access to device special files), `ST_NOEXEC` (disallow program execution), `ST_SYNCHRONOUS` (writes are synced at once), `ST_MANDLOCK` (allow mandatory locks on an FS), `ST_WRITE` (write on file/directory/symlink), `ST_APPEND` (append-only file), `ST_IMMUTABLE` (immutable file), `ST_NOATIME` (do not update access times), `ST_NODIRATIME` (do not update directory access times), `ST_RELATIME` (update atime relative to mtime/ctime).

This function can support *specifying a file descriptor*.

可用性: Unix.

在 3.2 版的變更: 新增 `ST_RDONLY` 與 `ST_NOSUID` 常數。

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor.

在 3.4 版的變更: The `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, and `ST_RELATIME` constants were added.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.7 版的變更: 新增 `f_fsid` 屬性。

os.supports_dir_fd

A *set* object indicating which functions in the *os* module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency's sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `None` for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression evaluates to `True` if *os.stat()* accepts open file descriptors for *dir_fd* on the local platform:

```
os.stat in os.supports_dir_fd
```

Currently *dir_fd* parameters only work on Unix platforms; none of them work on Windows.

在 3.3 版被加入.

os.supports_effective_ids

A *set* object indicating whether *os.access()* permits specifying `True` for its *effective_ids* parameter on the local platform. (Specifying `False` for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain *os.access()*; otherwise it will be empty.

This expression evaluates to `True` if *os.access()* supports `effective_ids=True` on the local platform:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* is only supported on Unix platforms; it does not work on Windows.

在 3.3 版被加入.

os.supports_fd

A *set* object indicating which functions in the *os* module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if *os.chdir()* accepts open file descriptors for *path* on your local platform:

```
os.chdir in os.supports_fd
```

在 3.3 版被加入.

os.supports_follow_symlinks

A *set* object indicating which functions in the *os* module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling *os.stat()* on the local platform:

```
os.stat in os.supports_follow_symlinks
```

在 3.3 版被加入.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

Create a symbolic link pointing to *src* named *dst*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is `True` or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

This function can support *paths relative to directory descriptors*.

備🄵

On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.

OSError is raised when the function is called by an unprivileged user.

引發一個附帶引數 `src`、`dst`、`dir_fd` 的稽核事件 `os.symlink`。

可用性: Unix, Windows.

The function is limited on WASI, see *WebAssembly 平台* for more information.

在 3.2 版的變更: Added support for Windows 6.0 (Vista) symbolic links.

在 3.3 版的變更: Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

在 3.6 版的變更: Accepts a *path-like object* for *src* and *dst*.

在 3.8 版的變更: Added support for unelevated symlinks on Windows with Developer Mode.

os.**sync**()

Force write of everything to disk.

可用性: Unix.

在 3.3 版被加入.

os.**truncate**(*path*, *length*)

Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.

This function can support *specifying a file descriptor*.

引發一個附帶引數 `path`、`length` 的稽核事件 `os.truncate`。

可用性: Unix, Windows.

在 3.3 版被加入.

在 3.5 版的變更: 新增對 Windows 的支援

在 3.6 版的變更: Accepts a *path-like object*.

os.**unlink**(*path*, ***, *dir_fd=None*)

Remove (delete) the file *path*. This function is semantically identical to *remove()*; the `unlink` name is its traditional Unix name. Please see the documentation for *remove()* for further information.

引發一個附帶引數 `path`、`dir_fd` 的稽核事件 `os.remove`。

在 3.3 版的變更: 新增 *dir_fd* 參數。

在 3.6 版的變更: Accepts a *path-like object*.

os.**utime**(*path*, *times=None*, *, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

Set the access and modified times of the file specified by *path*.

utime() takes two optional parameters, *times* and *ns*. These specify the times set on *path* and are used as follows:

- If *ns* is specified, it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- If *times* is not `None`, it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.
- If *times* is `None` and *ns* is unspecified, this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both *times* and *ns*.

Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the *os.stat()* result object with the *ns* parameter to *utime()*.

This function can support *specifying a file descriptor*, *paths relative to directory descriptors* and *not following symlinks*.

引發一個附帶引數 `path`、`times`、`ns`、`dir_fd` 的稽核事件 `os.utime`。

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

在 3.6 版的變更: Accepts a *path-like object*.

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

If optional argument *topdown* is `True` or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If *topdown* is `False`, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of *topdown*, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When *topdown* is `True`, the caller can modify the *dirnames* list in-place (perhaps using `del` or slice assignment), and *walk()* will only recurse into the subdirectories whose names remain in *dirnames*; this can be used to prune the search, impose a specific order of visiting, or even to inform *walk()* about directories the caller creates or renames before it resumes *walk()* again. Modifying *dirnames* when *topdown* is `False` has no effect on the behavior of the walk, because in bottom-up mode the directories in *dirnames* are generated before *dirpath* itself is generated.

By default, errors from the *scandir()* call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an *OSError* instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, *walk()* will not walk down into symbolic links that resolve to directories. Set *followlinks* to `True` to visit directories pointed to by symlinks, on systems that support them.

> **備🄵**
>
> Be aware that setting *followlinks* to `True` can lead to infinite recursion if a link points to a parent directory of itself. *walk()* does not keep track of the directories it visited already.

> **備🄵**
>
> If you pass a relative pathname, don't change the current working directory between resumptions of *walk()*. *walk()* never changes the current directory, and assumes that its caller doesn't either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any `__pycache__` subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

In the next example (simple implementation of *shutil.rmtree()*), walking the tree bottom-up is essential, *rmdir()* doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

引發一個附帶引數 `top`、`topdown`、`onerror`、`followlinks` 的稽核事件 `os.walk`。

在 3.5 版的變更: This function now calls *os.scandir()* instead of *os.listdir()*, making it faster by reducing the number of calls to *os.stat()*.

在 3.6 版的變更: Accepts a *path-like object*.

`os.`**`fwalk`**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

This behaves exactly like *walk()*, except that it yields a 4-tuple `(dirpath, dirnames, filenames, dirfd)`, and it supports `dir_fd`.

dirpath, *dirnames* and *filenames* are identical to *walk()* output, and *dirfd* is a file descriptor referring to the directory *dirpath*.

This function always supports *paths relative to directory descriptors* and *not following symlinks*. Note however that, unlike other functions, the *fwalk()* default value for *follow_symlinks* is `False`.

> **備🄵**
>
> Since *fwalk()* yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with *dup()*) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any __pycache__ subdirectory:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

In the next example, walking the tree bottom-up is essential: *rmdir()* doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

引發一個附帶引數 `top`、`topdown`、`onerror`、`follow_symlinks`、`dir_fd` 的稽核事件 `os.fwalk`。

可用性: Unix.

在 3.3 版被加入.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.7 版的變更: 新增對 *bytes* 路徑的支援。

os.**memfd_create**(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is *non-inheritable*.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

可用性: Linux >= 3.17 with glibc >= 2.27.

在 3.8 版被加入.

os.**MFD_CLOEXEC**

os.**MFD_ALLOW_SEALING**

os.**MFD_HUGETLB**

os.**MFD_HUGE_SHIFT**

os.**MFD_HUGE_MASK**

os.**MFD_HUGE_64KB**

os.**MFD_HUGE_512KB**

os.**MFD_HUGE_1MB**

os.**MFD_HUGE_2MB**

os.**MFD_HUGE_8MB**

os.**MFD_HUGE_16MB**

os.**MFD_HUGE_32MB**

os.**MFD_HUGE_256MB**

os.**MFD_HUGE_512MB**

os.**MFD_HUGE_1GB**

os.**MFD_HUGE_2GB**

os.**MFD_HUGE_16GB**

這些旗標可以傳給*memfd_create()*。

可用性: Linux >= 3.17 with glibc >= 2.27

MFD_HUGE* 旗標僅在 Linux 4.14 以上可用。

在 3.8 版被加入.

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw *read()* and *write()* with a buffer size of 8, *select()*, *poll()* and similar. See man page *eventfd(2)* for more information. By default, the new file descriptor is *non-inheritable*.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from *EFD_CLOEXEC*, *EFD_NONBLOCK*, and *EFD_SEMAPHORE*.

If *EFD_SEMAPHORE* is specified and the event counter is non-zero, *eventfd_read()* returns 1 and decrements the counter by one.

If *EFD_SEMAPHORE* is not specified and the event counter is non-zero, *eventfd_read()* returns the current event counter value and resets the counter to zero.

If the event counter is zero and *EFD_NONBLOCK* is not specified, *eventfd_read()* blocks.

eventfd_write() increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

範例:

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.10 版被加入.

os.**eventfd_read**(*fd*)

Read value from an *eventfd()* file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

可用性: Linux >= 2.6.27

在 3.10 版被加入.

os.**eventfd_write**(*fd*, *value*)

Add value to an *eventfd()* file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

可用性: Linux >= 2.6.27

在 3.10 版被加入.

os.**EFD_CLOEXEC**

Set close-on-exec flag for new *eventfd()* file descriptor.

可用性: Linux >= 2.6.27

在 3.10 版被加入.

os.**EFD_NONBLOCK**

設定新的*eventfd()* 檔案描述器的*O_NONBLOCK* 狀態旗標。

可用性: Linux >= 2.6.27

在 3.10 版被加入.

os.**EFD_SEMAPHORE**

Provide semaphore-like semantics for reads from an *eventfd()* file descriptor. On read the internal counter is decremented by one.

可用性: Linux >= 2.6.30

在 3.10 版被加入.

計時器檔案描述器

在 3.13 版被加入.

These functions provide support for Linux's *timer file descriptor* API. Naturally, they are all only available on Linux.

os.**timerfd_create**(*clockid*, */*, ***, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by *timerfd_create()* supports:

- *read()*
- *select()*
- *poll()*

The file descriptor's *read()* method can be called with a buffer size of 8. If the timer has already expired one or more times, *read()* returns the number of expirations with the host's endianness, which may be converted to an *int* by `int.from_bytes(x, byteorder=sys.byteorder)`.

select() and *poll()* can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid *clock ID*, as defined in the *time* module:

- *time.CLOCK_REALTIME*
- *time.CLOCK_MONOTONIC*
- *time.CLOCK_BOOTTIME* (Since Linux 3.15 for timerfd_create)

If *clockid* is *time.CLOCK_REALTIME*, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see *TFD_TIMER_CANCEL_ON_SET*.

If *clockid* is *time.CLOCK_MONOTONIC*, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is *time.CLOCK_BOOTTIME*, same as *time.CLOCK_MONOTONIC* except it includes any time that the system is suspended.

The file descriptor's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the | operator):

- *TFD_NONBLOCK*
- *TFD_CLOEXEC*

If *TFD_NONBLOCK* is not set as a flag, *read()* blocks until the timer expires. If it is set as a flag, *read()* doesn't block, but If there hasn't been an expiration since the last call to read, *read()* raises *OSError* with `errno` is set to *errno.EAGAIN*.

TFD_CLOEXEC 總是由 Python 自動設定。

The file descriptor must be closed with *os.close()* when it is no longer needed, or else the file descriptor will be leaked.

也參考

timerfd_create(2) 使用手🄵。

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

`os.`**`timerfd_settime`**(*fd*, */*, ***, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor's internal timer. This function operates the same interval timer as *timerfd_settime_ns()*.

fd 必須是有效的計時器檔案描述器。

The timer's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the | operator):

- *TFD_TIMER_ABSTIME*
- *TFD_TIMER_CANCEL_ON_SET*

The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an *OSError* exception with `errno` set to *errno.EINVAL*

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the *TFD_TIMER_ABSTIME* flag is set, the timer will fire when the timer's clock (set by *clockid* in *timerfd_create()*) reaches *initial* seconds.

The timer's interval is set by the *interval* *float*. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises *OSError* with `errno` set to *errno.EINVAL*

If the *TFD_TIMER_CANCEL_ON_SET* flag is set along with *TFD_TIMER_ABSTIME* and the clock for this timer is *time.CLOCK_REALTIME*, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn't cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events:

- `settimeofday`
- `clock_settime`
- set the system date and time by `date` command

Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

> **也參考**
>
> *timerfd_create(2)*, *timerfd_settime(2)*, *settimeofday(2)*, *clock_settime(2)*, and *date(1)*.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**timerfd_settime_ns**(*fd*, /, *, *flags=0*, *initial=0*, *interval=0*)

Similar to *timerfd_settime()*, but use time as nanoseconds. This function operates the same interval timer as *timerfd_settime()*.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**timerfd_gettime**(*fd*, /)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the *TFD_TIMER_ABSTIME* flag is set.

`interval` denotes the timer's interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

> **也參考**
>
> *timerfd_gettime(2)*

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**timerfd_gettime_ns**(*fd*, /)

Similar to *timerfd_gettime()*, but return time as nanoseconds.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**TFD_NONBLOCK**

A flag for the *timerfd_create()* function, which sets the *O_NONBLOCK* status flag for the new timer file descriptor. If *TFD_NONBLOCK* is not set as a flag, *read()* blocks.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**TFD_CLOEXEC**

A flag for the *timerfd_create()* function, If *TFD_CLOEXEC* is set as a flag, set close-on-exec flag for new file descriptor.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**TFD_TIMER_ABSTIME**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions. If this flag is set, *initial* is interpreted as an absolute value on the timer's clock (in UTC seconds or nanoseconds since the Unix Epoch).

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

os.**TFD_TIMER_CANCEL_ON_SET**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions along with *TFD_TIMER_ABSTIME*. The timer is cancelled when the time of the underlying clock changes discontinuously.

可用性: Linux >= 2.6.27 with glibc >= 2.8

在 3.13 版被加入.

Linux extended attributes

在 3.3 版被加入.

These functions are all available on Linux only.

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)

Return the value of the extended filesystem attribute *attribute* for *path*. *attribute* can be bytes or str (directly or indirectly through the *PathLike* interface). If it is str, it is encoded with the filesystem encoding.

This function can support *specifying a file descriptor* and *not following symlinks*.

引發一個附帶引數 `path`、`attribute` 的稽核事件 `os.getxattr`。

在 3.6 版的變更: Accepts a *path-like object* for *path* and *attribute*.

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)

Return a list of the extended filesystem attributes on *path*. The attributes in the list are represented as strings decoded with the filesystem encoding. If *path* is `None`, *listxattr()* will examine the current directory.

This function can support *specifying a file descriptor* and *not following symlinks*.

引發一個附帶引數 `path` 的稽核事件 `os.listxattr`。

在 3.6 版的變更: Accepts a *path-like object*.

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the *PathLike* interface). If it is a string, it is encoded with the *filesystem encoding and error handler*.

This function can support *specifying a file descriptor* and *not following symlinks*.

引發一個附帶引數 `path`、`attribute` 的稽核事件 `os.removexattr`。

在 3.6 版的變更: Accepts a *path-like object* for *path* and *attribute*.

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the *filesystem encoding and error handler*. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, `ENODATA` will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and `EEXISTS` will be raised.

This function can support *specifying a file descriptor* and *not following symlinks*.

> **備🄵**
>
> A bug in Linux kernel versions less than 2.6.39 caused the flags argument to be ignored on some filesystems.

引發一個附帶引數 `path`、`attribute`、`value`、`flags` 的稽核事件 `os.setxattr`。

在 3.6 版的變更: Accepts a *path-like object* for *path* and *attribute*.

os.**XATTR_SIZE_MAX**

The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

os.**XATTR_CREATE**

This is a possible value for the flags argument in *setxattr()*. It indicates the operation must create an attribute.

os.**XATTR_REPLACE**

This is a possible value for the flags argument in *setxattr()*. It indicates the operation must replace an existing attribute.

16.1.7 行程管理

These functions may be used to create and manage processes.

The various *exec** functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program's `main()`. For example, `os.execv('/bin/echo', ['foo', 'bar'])` will only print `bar` on standard output; `foo` will seem to be ignored.

os.**abort**()

Generate a `SIGABRT` signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of `3`. Be aware that calling this function will not call the Python signal handler registered for `SIGABRT` with *signal.signal()*.

os.**add_dll_directory**(*path*)

Add a path to the DLL search path.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through *sys.path*), and also by *ctypes*.

Remove the directory by calling **close()** on the returned object or using it in a `with` statement.

See the Microsoft documentation for more information about how DLLs are loaded.

引發一個附帶引數 `path` 的稽核事件 `os.add_dll_directory`。

可用性: Windows.

在 3.8 版被加入: Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching `PATH` or the current working directory, and OS functions such as `AddDllDirectory` having no effect.

In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

os.**execl**(*path*, *arg0*, *arg1*, ...)

os.**execle**(*path*, *arg0*, *arg1*, ..., *env*)

os.**execlp**(*file*, *arg0*, *arg1*, ...)

os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)

os.**execv**(*path*, *args*)

os.**execve**(*path*, *args*, *env*)

os.**execvp**(*file*, *args*)

os.**execvpe**(*file*, *args*, *env*)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as *OSError* exceptions.

The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or *os.fsync()* before calling an *exec** function.

The "l" and "v" variants of the *exec** functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written;

the individual parameters simply become additional parameters to the `execl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (*execlp()*, *execlpe()*, *execvp()*, and *execvpe()*) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the *exec*e* variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, *execl()*, *execle()*, *execv()*, and *execve()*, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

For *execle()*, *execlpe()*, *execve()*, and *execvpe()* (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process' environment); the functions *execl()*, *execlp()*, *execv()*, and *execvp()* all cause the new process to inherit the environment of the current process.

For *execve()* on some platforms, *path* may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using *os.supports_fd*. If it is unavailable, using it will raise a *NotImplementedError*.

引發一個附帶引數 `path`、`args`、`env` 的稽核事件 `os.exec`。

可用性: Unix, Windows, not WASI, not Android, not iOS.

在 3.3 版的變更: Added support for specifying *path* as an open file descriptor for *execve()*.

在 3.6 版的變更: Accepts a *path-like object*.

`os._exit(n)`

Exit the process with status *n*, without calling cleanup handlers, flushing stdio buffers, etc.

> **備🄵**
>
> The standard way to exit is *sys.exit(n)*. `_exit()` should normally only be used in the child process after a *fork()*.

The following exit codes are defined and can be used with *_exit()*, although they are not required. These are typically used for system programs written in Python, such as a mail server's external command delivery program.

> **備🄵**
>
> Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

`os.EX_OK`

Exit code that means no error occurred. May be taken from the defined value of `EXIT_SUCCESS` on some platforms. Generally has a value of zero.

可用性: Unix, Windows.

`os.EX_USAGE`

Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

可用性: Unix, not WASI.

`os.EX_DATAERR`

Exit code that means the input data was incorrect.

可用性: Unix, not WASI.

os.**EX_NOINPUT**

Exit code that means an input file did not exist or was not readable.

可用性: Unix, not WASI.

os.**EX_NOUSER**

Exit code that means a specified user did not exist.

可用性: Unix, not WASI.

os.**EX_NOHOST**

Exit code that means a specified host did not exist.

可用性: Unix, not WASI.

os.**EX_UNAVAILABLE**

Exit code that means that a required service is unavailable.

可用性: Unix, not WASI.

os.**EX_SOFTWARE**

Exit code that means an internal software error was detected.

可用性: Unix, not WASI.

os.**EX_OSERR**

Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

可用性: Unix, not WASI.

os.**EX_OSFILE**

Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

可用性: Unix, not WASI.

os.**EX_CANTCREAT**

Exit code that means a user specified output file could not be created.

可用性: Unix, not WASI.

os.**EX_IOERR**

Exit code that means that an error occurred while doing I/O on some file.

可用性: Unix, not WASI.

os.**EX_TEMPFAIL**

Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.

可用性: Unix, not WASI.

os.**EX_PROTOCOL**

Exit code that means that a protocol exchange was illegal, invalid, or not understood.

可用性: Unix, not WASI.

os.**EX_NOPERM**

Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

可用性: Unix, not WASI.

os.**EX_CONFIG**

Exit code that means that some kind of configuration error occurred.

可用性: Unix, not WASI.

os.**EX_NOTFOUND**

Exit code that means something like "an entry was not found".

可用性: Unix, not WASI.

os.**fork**()

Fork a child process. Return `0` in the child and the child's process id in the parent. If an error occurs *OSError* is raised.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using `fork()` from a thread.

引發一個不附帶引數的稽核事件 `os.fork`。

警告

If you use TLS sockets in an application calling `fork()`, see the warning in the *ssl* documentation.

警告

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

在 3.8 版的變更: Calling `fork()` in a subinterpreter is no longer supported (*RuntimeError* is raised).

在 3.12 版的變更: If Python is able to detect that your process has multiple threads, *os.fork()* now raises a *DeprecationWarning*.

We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with *os.fork()* on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).

Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.

See this discussion on fork being incompatible with threads for technical details of why we're surfacing this longstanding platform compatibility problem to developers.

可用性: POSIX, not WASI, not Android, not iOS.

os.**forkpty**()

Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of `(pid, fd)`, where *pid* is `0` in the child, the new child's process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the *pty* module. If an error occurs *OSError* is raised.

引發一個不附帶引數的稽核事件 `os.forkpty`。

警告

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

在 3.8 版的變更: Calling `forkpty()` in a subinterpreter is no longer supported (*RuntimeError* is raised).

在 3.12 版的變更: If Python is able to detect that your process has multiple threads, this now raises a *DeprecationWarning*. See the longer explanation on *os.fork()*.

可用性: Unix, not WASI, not Android, not iOS.

os.**kill**(*pid*, *sig*, */*)

Send signal *sig* to the process *pid*. Constants for the specific signals available on the host platform are defined in the *signal* module.

Windows: The *signal.CTRL_C_EVENT* and *signal.CTRL_BREAK_EVENT* signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*.

另請參🄵*signal.pthread_kill()*。

引發一個附帶引數 `pid`、`sig` 的稽核事件 `os.kill`。

可用性: Unix, Windows, not WASI, not iOS.

在 3.2 版的變更: 新支援 Windows。

os.**killpg**(*pgid*, *sig*, */*)

Send the signal *sig* to the process group *pgid*.

引發一個附帶引數 `pgid`、`sig` 的稽核事件 `os.killpg`。

可用性: Unix, not WASI, not iOS.

os.**nice**(*increment*, */*)

Add *increment* to the process's "niceness". Return the new niceness.

可用性: Unix, not WASI.

os.**pidfd_open**(*pid*, *flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

更多細節請見 *pidfd_open(2)* 使用手🄵。

可用性: Linux >= 5.3, Android >= *build-time* API level 31

在 3.9 版被加入.

os.**PIDFD_NONBLOCK**

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid(2)* will immediately return the error *EAGAIN* rather than blocking.

可用性: Linux >= 5.10

在 3.12 版被加入.

os.**plock**(*op*, */*)

Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked.

可用性: Unix, not WASI, not iOS.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument have the same meaning as the corresponding argument to the built-in *open()* function. The returned file object reads or writes text strings rather than bytes.

The `close` method returns *None* if the subprocess exited successfully, or the subprocess's return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be `- signal.SIGKILL` if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

On Unix, *waitstatus_to_exitcode()* can be used to convert the `close` method result (exit status) into an exit code if it is not `None`. On Windows, the `close` method result is directly the exit code (or `None`).

This is implemented using *subprocess.Popen*; see that class's documentation for more powerful ways to manage and communicate with subprocesses.

可用性: not WASI, not Android, not iOS.

> **備🄵**
>
> The *Python UTF-8 Mode* affects encodings used for *cmd* and pipe contents.
>
> *popen()* is a simple wrapper around *subprocess.Popen*. Use *subprocess.Popen* or *subprocess.run()* to control options like encodings.

在 3.14 版之後被🄴用: The function is *soft deprecated* and should no longer be used to write new code. The *subprocess* module is recommended instead.

`os.`**`posix_spawn`**(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawn()` C library API for use from Python.

Most users should use *subprocess.run()* instead of *posix_spawn()*.

The positional-only arguments *path*, *args*, and *env* are similar to *execve()*. *env* is allowed to be `None`, in which case current process' environment is used.

The *path* parameter is the path to the executable file. The *path* should contain a directory. Use *posix_spawnp()* to pass an executable file without directory.

The *file_actions* argument may be a sequence of tuples describing actions to take on specific file descriptors in the child process between the C library implementation's `fork()` and `exec()` steps. The first item in each tuple must be one of the three type indicator listed below describing the remaining tuple elements:

`os.`**`POSIX_SPAWN_OPEN`**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

Performs `os.dup2(os.open(path, flags, mode), fd)`.

`os.`**`POSIX_SPAWN_CLOSE`**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

Performs `os.close(fd)`.

`os.`**`POSIX_SPAWN_DUP2`**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

Performs `os.dup2(fd, new_fd)`.

`os.`**`POSIX_SPAWN_CLOSEFROM`**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()`, and `posix_spawn_file_actions_addclosefrom_np()` API calls used to prepare for the `posix_spawn()` call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child's process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent's process group ID. This argument corresponds to the C library `POSIX_SPAWN_SETPGROUP` flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library `POSIX_SPAWN_RESETIDS` flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires `POSIX_SPAWN_SETSID` or `POSIX_SPAWN_SETSID_NP` flag. Otherwise, *NotImplementedError* is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent's signal mask. This argument corresponds to the C library `POSIX_SPAWN_SETSIGMASK` flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library `POSIX_SPAWN_SETSIGDEF` flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of *sched_param* with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library `POSIX_SPAWN_SETSCHEDPARAM` and `POSIX_SPAWN_SETSCHEDULER` flags.

引發一個附帶引數 `path`、`argv`、`env` 的稽核事件 `os.posix_spawn`。

在 3.8 版被加入.

在 3.13 版的變更: *env* parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where `posix_spawn_file_actions_addclosefrom_np()` exists.

可用性: Unix, not WASI, not Android, not iOS.

`os.`**`posix_spawnp`**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawnp()` C library API for use from Python.

Similar to *posix_spawn()* except that the system searches for the *executable* file in the list of directories specified by the `PATH` environment variable (in the same way as for `execvp(3)`).

引發一個附帶引數 `path`、`argv`、`env` 的稽核事件 `os.posix_spawn`。

在 3.8 版被加入.

可用性: POSIX, not WASI, not Android, not iOS.

見*posix_spawn()* 文件。

`os.`**`register_at_fork`**(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Register callables to be executed when a new child process is forked using *os.fork()* or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- *before* is a function called before forking a child process.
- *after_in_parent* is a function called from the parent process after forking a child process.
- *after_in_child* is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical *subprocess* launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

可用性: Unix, not WASI, not Android, not iOS.

在 3.7 版被加入.

os.**spawnl**(*mode*, *path*, *...*)

os.**spawnle**(*mode*, *path*, *...*, *env*)

os.**spawnlp**(*mode*, *file*, *...*)

os.**spawnlpe**(*mode*, *file*, *...*, *env*)

os.**spawnv**(*mode*, *path*, *args*)

os.**spawnve**(*mode*, *path*, *args*, *env*)

os.**spawnvp**(*mode*, *file*, *args*)

os.**spawnvpe**(*mode*, *file*, *args*, *env*)

Execute the program *path* in a new process.

(Note that the *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the *Replacing Older Functions with the subprocess Module* section.)

If *mode* is *P_NOWAIT*, this function returns the process id of the new process; if *mode* is *P_WAIT*, returns the process's exit code if it exits normally, or `-signal`, where *signal* is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the *waitpid()* function.

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

The "l" and "v" variants of the *spawn** functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (*spawnlp()*, *spawnlpe()*, *spawnvp()*, and *spawnvpe()*) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the *spawn*e* variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, *spawnl()*, *spawnle()*, *spawnv()*, and *spawnve()*, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For *spawnle()*, *spawnlpe()*, *spawnve()*, and *spawnvpe()* (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process' environment); the functions *spawnl()*, *spawnlp()*, *spawnv()*, and *spawnvp()* all cause the new process to inherit the environment of the current process. Note that keys and values in the *env* dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of `127`.

As an example, the following calls to *spawnlp()* and *spawnvpe()* are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

引發一個附帶引數 `mode`、`path`、`args`、`env` 的稽核事件 `os.spawn`。

可用性: Unix, Windows, not WASI, not Android, not iOS.

spawnlp(), *spawnlpe()*, *spawnvp()* and *spawnvpe()* are not available on Windows. *spawnle()* and *spawnve()* are not thread-safe on Windows; we advise you to use the *subprocess* module instead.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.14 版之後被🄴用: These functions are *soft deprecated* and should no longer be used to write new code. The *subprocess* module is recommended instead.

os.**P_NOWAIT**

os.**P_NOWAITO**

Possible values for the *mode* parameter to the *spawn** family of functions. If either of these values is given, the *spawn** functions will return as soon as the new process has been created, with the process id as the return value.

可用性: Unix, Windows.

os.**P_WAIT**

Possible value for the *mode* parameter to the *spawn** family of functions. If this is given as *mode*, the *spawn** functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

可用性: Unix, Windows.

os.**P_DETACH**

os.**P_OVERLAY**

Possible values for the *mode* parameter to the *spawn** family of functions. These are less portable than those listed above. *P_DETACH* is similar to *P_NOWAIT*, but the new process is detached from the console of the calling process. If *P_OVERLAY* is used, the current process will be replaced; the *spawn** function will not return.

可用性: Windows.

os.**startfile**(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

Start a file with its associated application.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 `ShellExecute()` function.

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use *pathlib* or the *os.path.normpath()* function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 `ShellExecute()` function is not resolved until this function is first called. If the function cannot be resolved, *NotImplementedError* will be raised.

引發一個附帶引數 `path`、`operation` 的稽核事件 `os.startfile`。

引發一個附帶引數 `path`、`operation`、`arguments`、`cwd`、`show_cmd` 的稽核事件 `os.startfile/2`。

可用性: Windows.

在 3.10 版的變更: Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

os.**system**(*command*)

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to *sys.stdin*, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for *wait()*.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable COMSPEC: it is usually **cmd.exe**, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results; using that module is recommended to using this function. See the *Replacing Older Functions with the subprocess Module* section in the *subprocess* documentation for some helpful recipes.

On Unix, *waitstatus_to_exitcode()* can be used to convert the result (exit status) into an exit code. On Windows, the result is directly the exit code.

引發一個附帶引數 command 的稽核事件 os.system。

可用性: Unix, Windows, not WASI, not Android, not iOS.

os.**times**()

Returns the current global process times. The return value is an object with five attributes:

- user - 使用者時間
- system - 系統時間
- children_user - 所有子行程的使用者時間
- children_system - 所有子行程的系統時間
- elapsed - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing user, system, children_user, children_system, and elapsed in that order.

See the Unix manual page *times(2)* and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only user and system are known; the other attributes are zero.

可用性: Unix, Windows.

在 3.3 版的變更: Return type changed from a tuple to a tuple-like object with named attributes.

os.**wait**()

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

If there are no children that could be waited for, *ChildProcessError* is raised.

waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

可用性: Unix, not WASI, not Android, not iOS.

也參考

The other wait*() functions documented below can be used to wait for the completion of a specific child process and have more options. *waitpid()* is the only one also available on Windows.

os.**waitid**(*idtype*, *id*, *options*, */*)

Wait for the completion of a child process.

idtype can be *P_PID*, *P_PGID*, *P_ALL*, or (on Linux) *P_PIDFD*. The interpretation of *id* depends on it; see their individual descriptions.

options is an OR combination of flags. At least one of *WEXITED*, *WSTOPPED* or *WCONTINUED* is required; *WNOHANG* and *WNOWAIT* are additional optional flags.

The return value is an object representing the data contained in the siginfo_t structure with the following attributes:

- si_pid (process ID)
- si_uid (real user ID of the child)
- si_signo (always *SIGCHLD*)
- si_status (the exit status or signal number, depending on si_code)
- si_code (see *CLD_EXITED* for possible values)

If *WNOHANG* is specified and there are no matching children in the requested state, None is returned. Otherwise, if there are no matching children that could be waited for, *ChildProcessError* is raised.

可用性: Unix, not WASI, not Android, not iOS.

在 3.3 版被加入.

在 3.13 版的變更: This function is now available on macOS as well.

os.**waitpid**(*pid*, *options*, /)

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for *wait()*). The semantics of the call are affected by the value of the integer *options*, which should be 0 for normal operation.

If *pid* is greater than 0, *waitpid()* requests status information for that specific process. If *pid* is 0, the request is for the status of any child in the process group of the current process. If *pid* is -1, the request pertains to any child of the current process. If *pid* is less than -1, status is requested for any process in the process group -*pid* (the absolute value of *pid*).

options is an OR combination of flags. If it contains *WNOHANG* and there are no matching children in the requested state, (0, 0) is returned. Otherwise, if there are no matching children that could be waited for, *ChildProcessError* is raised. Other options that can be used are *WUNTRACED* and *WCONTINUED*.

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to 0 has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The *spawn** functions called with *P_NOWAIT* return suitable process handles.

waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

可用性: Unix, Windows, not WASI, not Android, not iOS.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

os.**wait3**(*options*)

Similar to *waitpid()*, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to *resource.getrusage()* for details on resource usage information. The *options* argument is the same as that provided to *waitpid()* and *wait4()*.

waitstatus_to_exitcode() can be used to convert the exit status into an exitcode.

可用性: Unix, not WASI, not Android, not iOS.

os.**wait4**(*pid*, *options*)

Similar to *waitpid()*, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to *resource.getrusage()* for details on resource usage information. The arguments to *wait4()* are the same as those provided to *waitpid()*.

waitstatus_to_exitcode() can be used to convert the exit status into an exitcode.

可用性: Unix, not WASI, not Android, not iOS.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

os.**P_PIDFD**

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted:

- `P_PID` - wait for the child whose PID is *id*.
- `P_PGID` - wait for any child whose progress group ID is *id*.
- `P_ALL` - wait for any child; *id* is ignored.
- `P_PIDFD` - wait for the child identified by the file descriptor *id* (a process file descriptor created with `pidfd_open()`).

可用性: Unix, not WASI, not Android, not iOS.

> **備🄵**
>
> `P_PIDFD` is only available on Linux >= 5.4.

在 3.3 版被加入.

在 3.9 版被加入: The `P_PIDFD` constant.

os.**WCONTINUED**

This *options* flag for `waitpid()`, `wait3()`, `wait4()`, and `waitid()` causes child processes to be reported if they have been continued from a job control stop since they were last reported.

可用性: Unix, not WASI, not Android, not iOS.

os.**WEXITED**

This *options* flag for `waitid()` causes child processes that have terminated to be reported.

The other `wait*` functions always report children that have terminated, so this option is not available for them.

可用性: Unix, not WASI, not Android, not iOS.

在 3.3 版被加入.

os.**WSTOPPED**

This *options* flag for `waitid()` causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

可用性: Unix, not WASI, not Android, not iOS.

在 3.3 版被加入.

os.**WUNTRACED**

This *options* flag for `waitpid()`, `wait3()`, and `wait4()` causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for `waitid()`.

可用性: Unix, not WASI, not Android, not iOS.

os.**WNOHANG**

This *options* flag causes `waitpid()`, `wait3()`, `wait4()`, and `waitid()` to return right away if no child process status is available immediately.

可用性: Unix, not WASI, not Android, not iOS.

os.**WNOWAIT**

This *options* flag causes *waitid()* to leave the child in a waitable state, so that a later `wait*()` call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

可用性: Unix, not WASI, not Android, not iOS.

os.**CLD_EXITED**

os.**CLD_KILLED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_STOPPED**

os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by *waitid()*.

可用性: Unix, not WASI, not Android, not iOS.

在 3.3 版被加入.

在 3.9 版的變更: 新增*CLD_KILLED* 和*CLD_STOPPED* 值。

os.**waitstatus_to_exitcode**(*status*)

Convert a wait status to an exit code.

在 Unix 上:

- If the process exited normally (if `WIFEXITED(status)` is true), return the process exit status (return `WEXITSTATUS(status)`): result greater than or equal to 0.
- If the process was terminated by a signal (if `WIFSIGNALED(status)` is true), return `-signum` where *signum* is the number of the signal that caused the process to terminate (return `-WTERMSIG(status)`): result less than 0.
- 否則，引發*ValueError*。

On Windows, return *status* shifted right by 8 bits.

On Unix, if the process is being traced or if *waitpid()* was called with *WUNTRACED* option, the caller must first check if `WIFSTOPPED(status)` is true. This function must not be called if `WIFSTOPPED(status)` is true.

也參考

WIFEXITED(), *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()* functions.

可用性: Unix, Windows, not WASI, not Android, not iOS.

在 3.9 版被加入.

The following functions take a process status code as returned by *system()*, *wait()*, or *waitpid()* as a parameter. They may be used to determine the disposition of a process.

os.**WCOREDUMP**(*status*, /)

Return `True` if a core dump was generated for the process, otherwise return `False`.

This function should be employed only if *WIFSIGNALED()* is true.

可用性: Unix, not WASI, not Android, not iOS.

os.**WIFCONTINUED**(*status*)

Return `True` if a stopped child has been resumed by delivery of *SIGCONT* (if the process has been continued from a job control stop), otherwise return `False`.

參🄵*WCONTINUED* 選項。

可用性: Unix, not WASI, not Android, not iOS.

os.**WIFSTOPPED**(*status*)

Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

WIFSTOPPED() only returns `True` if the *waitpid()* call was done using *WUNTRACED* option or when the process is being traced (see *ptrace(2)*).

可用性: Unix, not WASI, not Android, not iOS.

os.**WIFSIGNALED**(*status*)

Return `True` if the process was terminated by a signal, otherwise return `False`.

可用性: Unix, not WASI, not Android, not iOS.

os.**WIFEXITED**(*status*)

Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`; otherwise return `False`.

可用性: Unix, not WASI, not Android, not iOS.

os.**WEXITSTATUS**(*status*)

Return the process exit status.

This function should be employed only if *WIFEXITED()* is true.

可用性: Unix, not WASI, not Android, not iOS.

os.**WSTOPSIG**(*status*)

Return the signal which caused the process to stop.

This function should be employed only if *WIFSTOPPED()* is true.

可用性: Unix, not WASI, not Android, not iOS.

os.**WTERMSIG**(*status*)

Return the number of the signal that caused the process to terminate.

This function should be employed only if *WIFSIGNALED()* is true.

可用性: Unix, not WASI, not Android, not iOS.

16.1.8 排程器介面

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages.

在 3.3 版被加入.

The following scheduling policies are exposed if they are supported by the operating system.

os.**SCHED_OTHER**

The default scheduling policy.

os.**SCHED_BATCH**

Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

os.**SCHED_DEADLINE**

Scheduling policy for tasks with deadline constraints.

在 3.14 版被加入.

os.**SCHED_IDLE**

Scheduling policy for extremely low priority background tasks.

os.**SCHED_NORMAL**

Alias for *SCHED_OTHER*.

在 3.14 版被加入.

os.**SCHED_SPORADIC**

Scheduling policy for sporadic server programs.

os.**SCHED_FIFO**

先進先出排程策略。

os.**SCHED_RR**

輪轉排程策略。

os.**SCHED_RESET_ON_FORK**

This flag can be OR'ed with any other scheduling policy. When a process with this flag set forks, its child's scheduling policy and priority are reset to the default.

class os.**sched_param**(*sched_priority*)

This class represents tunable scheduling parameters used in *sched_setparam()*, *sched_setscheduler()*, and *sched_getparam()*. It is immutable.

At the moment, there is only one possible parameter:

sched_priority

The scheduling priority for a scheduling policy.

os.**sched_get_priority_min**(*policy*)

Get the minimum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_get_priority_max**(*policy*)

Get the maximum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_setscheduler**(*pid*, *policy*, *param*, */*)

Set the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. *policy* is one of the scheduling policy constants above. *param* is a *sched_param* instance.

os.**sched_getscheduler**(*pid*, */*)

Return the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. The result is one of the scheduling policy constants above.

os.**sched_setparam**(*pid*, *param*, */*)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a *sched_param* instance.

os.**sched_getparam**(*pid*, */*)

Return the scheduling parameters as a *sched_param* instance for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_rr_get_interval**(*pid*, */*)

Return the round-robin quantum in seconds for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_yield**()

Voluntarily relinquish the CPU. See *sched_yield(2)* for details.

os.**sched_setaffinity**(*pid*, *mask*, */*)

Restrict the process with PID *pid* (or the current process if zero) to a set of CPUs. *mask* is an iterable of integers representing the set of CPUs to which the process should be restricted.

os.**sched_getaffinity**(*pid*, /)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

也請見*process_cpu_count()* 函式。

16.1.9 Miscellaneous System Information

os.**confstr**(*name*, /)

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn't defined, `None` is returned.

If *name* is a string and is not known, *ValueError* is raised. If a specific value for *name* is not supported by the host system, even if it is included in `confstr_names`, an *OSError* is raised with *errno.EINVAL* for the error number.

可用性: Unix.

os.**confstr_names**

Dictionary mapping names accepted by *confstr()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

可用性: Unix.

os.**cpu_count**()

Return the number of logical CPUs in the **system**. Returns `None` if undetermined.

The *process_cpu_count()* function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

在 3.4 版被加入.

在 3.13 版的變更: If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, *cpu_count()* returns the overridden value *n*.

os.**getloadavg**()

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises *OSError* if the load average was unobtainable.

可用性: Unix.

os.**process_cpu_count**()

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns `None` if undetermined. It can be less than *cpu_count()* depending on the CPU affinity.

The *cpu_count()* function can be used to get the number of logical CPUs in the **system**.

If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, *process_cpu_count()* returns the overridden value *n*.

也請見*sched_getaffinity()* 函式。

在 3.13 版被加入.

os.**sysconf**(*name*, /)

Return integer-valued system configuration values. If the configuration value specified by *name* isn't defined, `-1` is returned. The comments regarding the *name* parameter for *confstr()* apply here as well; the dictionary that provides information on the known names is given by `sysconf_names`.

可用性: Unix.

os.**sysconf_names**

Dictionary mapping names accepted by *sysconf()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

可用性: Unix.

在 3.11 版的變更: Add `'SC_MINSIGSTKSZ'` name.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the *os.path* module.

os.**curdir**

The constant string used by the operating system to refer to the current directory. This is `'.'` for Windows and POSIX. Also available via *os.path*.

os.**pardir**

The constant string used by the operating system to refer to the parent directory. This is `'..'` for Windows and POSIX. Also available via *os.path*.

os.**sep**

The character used by the operating system to separate pathname components. This is `'/'` for POSIX and `'\\'` for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames --- use *os.path.split()* and *os.path.join()* --- but it is occasionally useful. Also available via *os.path*.

os.**altsep**

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `'/'` on Windows systems where `sep` is a backslash. Also available via *os.path*.

os.**extsep**

The character which separates the base filename from the extension; for example, the `'.'` in `os.py`. Also available via *os.path*.

os.**pathsep**

The character conventionally used by the operating system to separate search path components (as in `PATH`), such as `':'` for POSIX or `';'` for Windows. Also available via *os.path*.

os.**defpath**

The default search path used by *exec*p** and *spawn*p** if the environment doesn't have a `'PATH'` key. Also available via *os.path*.

os.**linesep**

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as `'\n'` for POSIX, or multiple characters, for example, `'\r\n'` for Windows. Do not use *os.linesep* as a line terminator when writing files opened in text mode (the default); use a single `'\n'` instead, on all platforms.

os.**devnull**

The file path of the null device. For example: `'/dev/null'` for POSIX, `'nul'` for Windows. Also available via *os.path*.

os.**RTLD_LAZY**

os.**RTLD_NOW**

os.**RTLD_GLOBAL**

os.**RTLD_LOCAL**

os.**RTLD_NODELETE**

os.**RTLD_NOLOAD**

os.**RTLD_DEEPBIND**

Flags for use with the *setdlopenflags()* and *getdlopenflags()* functions. See the Unix manual page *dlopen(3)* for what the different flags mean.

在 3.3 版被加入.

16.1.10 Random numbers

os.**getrandom**(*size*, *flags=0*)

Get up to *size* random bytes. The function can return less bytes than requested.

These bytes can be used to seed user-space random number generators or for cryptographic purposes.

`getrandom()` relies on entropy gathered from device drivers and other sources of environmental noise. Unnecessarily reading large quantities of data will have a negative impact on other users of the `/dev/random` and `/dev/urandom` devices.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: *os.GRND_RANDOM* and *GRND_NONBLOCK*.

See also the Linux getrandom() manual page.

可用性: Linux >= 3.17.

在 3.6 版被加入.

os.**urandom**(*size*, */*)

Return a bytestring of *size* random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation.

On Linux, if the `getrandom()` syscall is available, it is used in blocking mode: block until the system urandom entropy pool is initialized (128 bits of entropy are collected by the kernel). See the **PEP 524** for the rationale. On Linux, the *getrandom()* function can be used to get random bytes in non-blocking mode (using the *GRND_NONBLOCK* flag) or to poll until the system urandom entropy pool is initialized.

On a Unix-like system, random bytes are read from the `/dev/urandom` device. If the `/dev/urandom` device is not available or not readable, the *NotImplementedError* exception is raised.

On Windows, it will use `BCryptGenRandom()`.

> **也參考**
>
> The *secrets* module provides higher level functions. For an easy-to-use interface to the random number generator provided by your platform, please see *random.SystemRandom*.

在 3.5 版的變更: On Linux 3.17 and newer, the `getrandom()` syscall is now used when available. On OpenBSD 5.6 and newer, the C `getentropy()` function is now used. These functions avoid the usage of an internal file descriptor.

在 3.5.2 版的變更: On Linux, if the `getrandom()` syscall blocks (the urandom entropy pool is not initialized yet), fall back on reading `/dev/urandom`.

在 3.6 版的變更: On Linux, `getrandom()` is now used in blocking mode to increase the security.

在 3.11 版的變更: On Windows, `BCryptGenRandom()` is used instead of `CryptGenRandom()` which is deprecated.

os.**GRND_NONBLOCK**

By default, when reading from `/dev/random`, *getrandom()* blocks if no random bytes are available, and when reading from `/dev/urandom`, it blocks if the entropy pool has not yet been initialized.

If the *GRND_NONBLOCK* flag is set, then *getrandom()* does not block in these cases, but instead immediately raises *BlockingIOError*.

在 3.6 版被加入.

os.**GRND_RANDOM**

If this bit is set, then random bytes are drawn from the `/dev/random` pool instead of the `/dev/urandom` pool.

在 3.6 版被加入.

16.2 io —處理資料串流的核心工具

原始碼： Lib/io.py

16.2.1 總覽

io 模組替 Python 提供處理各種類型 IO 的主要工具。有三種主要的 IO 類型：*文字 I/O (text I/O)*、*二進位 I/O (binary I/O)* 以及*原始 I/O (raw I/O)*。這些均🄵泛用 (generic) 類型，且每種類型都可以使用各式後端儲存 (backing store)。任一種屬於這些類型的具體物件稱🄵*file object*。其它常見的名詞還有*資料串流 (stream)* 以及*類檔案物件 (file-like objects)*。

無論其類型🄵何，每個具體的資料串流物件也將具有各種能力：唯讀的、只接受寫入的、或者讀寫兼具的。它還允許任意的隨機存取（向前或向後尋找至任意位置），或者只能依順序存取（例如 socket 或 pipe 的情形下）。

所有的資料串流都會謹慎處理你所提供的資料的型🄵。舉例來🄵，提供一個 *str* 物件給二進位資料串流的 `write()` 方法將會引發 *TypeError*。同樣地，若提供一個 *bytes* 物件給文字資料串流的 `write()` 方法，也會引發同樣的錯誤。

在 3.3 版的變更: 原本會引發 *IOError* 的操作，現在將改成引發 *OSError*。因🄵 *IOError* 現在是 *OSError* 的🄵名。

文字 I/O

文字 I/O 要求和🄵出 *str* 物件。這意味著每當後端儲存🄵原生 bytes 時（例如在檔案的情形下），資料的編碼與解碼會以清楚易懂的方式進行，也可選擇同時轉🄵特定於平台的🄵行字元。

建立文字資料串流最簡單的方法是使用 *open()*，可選擇性地指定編碼:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

記憶體🄵的文字資料串流也可以使用 *StringIO* 物件建立:

```
f = io.StringIO("some initial text data")
```

> **備🄵**
>
> When working with a non-blocking stream, be aware that read operations on text I/O objects might raise a *BlockingIOError* if the stream cannot perform the operation immediately.

文字資料串流 API 的詳細🄵明在 *TextIOBase* 文件當中。

二進位 (Binary) I/O

二進位 I/O（也稱🄵緩衝 *I/O (buffered I/O)*）要求的是*類位元組物件 (bytes-like objects)* 且🄵生 *bytes* 物件。不進行編碼、解碼或者🄵行字元轉🄵。這種類型的資料串流可用於各種非文字資料，以及需要手動控制對文字資料的處理時。

建立二進位資料串流最簡單的方法是使用 *open()*，🄵在 mode 字串中加入 `'b'`:

```
f = open("myfile.jpg", "rb")
```

記憶體🄵的二進位資料串流也可以透過 *BytesIO* 物件來建立:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

二進位資料串流 API 的詳細🄵明在*BufferedIOBase* 文件當中。

其它函式庫模組可能提供額外的方法來建立文字或二進位資料串流。例如*socket.socket.makefile()*。

原始 (Raw) I/O

原始 I/O（也稱🄵無緩衝 *I/O (unbuffered I/O)*）通常作🄵二進位以及文字資料串流的低階 building-block 使用；在使用者程式碼中直接操作原始資料串流很少有用。然而，你可以透過以無緩衝的二進位模式開🄵一個檔案來建立一個原始資料串流：

```
f = open("myfile.jpg", "rb", buffering=0)
```

原始串流 API 在*RawIOBase* 文件中有詳細描述。

16.2.2 文字編碼

TextIOWrapper 和*open()* 預設編碼是根據區域設定的 (locale-specific) (*locale.getencoding()*)。

然而，許多開發人員在開🄵以 UTF-8 編碼的文字檔案（例如：JSON、TOML、Markdown 等）時忘記指定編碼，因🄵多數 Unix 平台預設使用 UTF-8 區域設定。這會導致錯誤，因🄵對於大多數 Windows 使用者來🄵，預設地區編碼🄵非 UTF-8。舉例來🄵：

```
# May not work on Windows when non-ASCII characters in the file.
with open("README.md") as f:
    long_description = f.read()
```

因此，🄵烈建議在開🄵文字檔案時，明確指定編碼。若你想使用 UTF-8 編碼，請傳入 `encoding="utf-8"`。若想使用目前的地區編碼，Python 3.10 以後的版本支援使用 `encoding="locale"`。

也參考

Python UTF-8 模式
在 Python UTF-8 模式下，可以將預設編碼從特定地區編碼改🄵 UTF-8。

PEP 686
Python 3.15 將預設使用*Python UTF-8 模式*。

選擇性加入的編碼警告

在 3.10 版被加入: 更多資訊請見 **PEP 597**。

要找出哪些地方使用到預設的地區編碼，你可以🄵用 `-X warn_default_encoding` 命令列選項，或者設定環境變數 `PYTHONWARNDEFAULTENCODING`。當使用到預設編碼時，會引發*EncodingWarning*。

如果你正在提供一個使用*open()* 或*TextIOWrapper* 且傳遞 `encoding=None` 作🄵參數的 API，你可以使用*text_encoding()*。如此一來如果 API 的呼叫方🄵有傳遞 `encoding`，呼叫方就會發出一個*EncodingWarning*。然而，對於新的 API，請考慮預設使用 UTF-8（即 `encoding="utf-8"`）。

16.2.3 高階模組介面

`io.`**`DEFAULT_BUFFER_SIZE`**

一個包含模組中緩衝 I/O 類🄵所使用的預設緩衝區大小的整數。若可能的話，*open()* 會使用檔案的 blksize （透過*os.stat()* 取得）。

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

這是🄵建函式*open()* 的🄵名。

此函式會引發一個帶有引數 *path*、*mode* 以及 *flags* 的稽核事件 *(auditing event)* `open`。*mode* 與 *flags* 引數可能已經被修改或者從原始呼叫中被推斷出來。

io.**open_code**(*path*)

以 'rb' 模式開🄵提供的檔案。此函式應用於意圖將🄵容視🄵可執行的程式碼的情🄵下。

path 應該要屬於 *str* 類🄵，且是個🄵對路徑。

這個函式的行🄵可能會被之前對 PyFile_SetOpenCodeHook() 的呼叫覆寫。然而，假設 *path* 是個 *str* 且🄵🄵對路徑，則 open_code(path) 總是與 open(path, 'rb') 有相同行🄵。覆寫這個行🄵是🄵了對檔案進行額外驗證或預處理。

在 3.8 版被加入.

io.**text_encoding**(*encoding*, *stacklevel=2*, */*)

這是個輔助函式，適用於使用 *open()* 或 *TextIOWrapper* 且具有 encoding=None 參數的可呼叫物件。

若 *encoding* 不🄵 None，此函式將回傳 *encoding*。否則，將根據 *UTF-8 Mode* 回傳 "locale" 或 "utf-8"。

若 *sys.flags.warn_default_encoding* 🄵 真，且 *encoding* 🄵 None，此函式會發出一個 *EncodingWarning*。*stacklevel* 指定警告在哪層發出。範例：

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

在此範例中，對於 read_text() 的呼叫方會引發一個 *EncodingWarning*。

更多資訊請見 文字編碼。

在 3.10 版被加入.

在 3.11 版的變更: 當 UTF-8 模式🄵用且 *encoding* 🄵 None 時，*text_encoding()* 會回傳"utf-8"。

exception io.**BlockingIOError**

這是🄵建的 *BlockingIOError* 例外的相容性🄵名。

exception io.**UnsupportedOperation**

當在資料串流上呼叫不支援的操作時，會引發繼承自 *OSError* 與 *ValueError* 的例外。

也參考

sys
包含標準的 IO 資料串流：*sys.stdin*、*sys.stdout* 以及 *sys.stderr*。

16.2.4 類🄵階層

I/O 串流的實作是由多個類🄵組合成的階層結構所構成。首先是 *abstract base classes*（抽象基底類🄵，ABCs），它們被用來規範各種不同類型的串流，接著具體類🄵會提供標準串流的實作。

備🄵

🄵了協助具體串流類🄵的實作，抽象基底類🄵提供了某些方法的預設實作。舉例來🄵，*BufferedIOBase* 提供未經最佳化的 readinto() 與 readline() 實作。

I/O 階層結構的最上層是抽象基底類🄵 *IOBase*。它定義了串流的基礎的介面。然而，請注意，讀取串流與寫入串流之間🄵🄵有分離；若不支援給定的操作，實作是允許引發 *UnsupportedOperation* 例外的。

抽象基底類🄵 *RawIOBase* 繼承 *IOBase*。此類🄵處理對串流的位元組讀寫。*FileIO* 則繼承 *RawIOBase* 來提供一個介面以存取機器檔案系統🄵的檔案。

抽象基底類🄵 *BufferedIOBase* 繼承 *IOBase*。此類🄵緩衝原始二進位串流 (*RawIOBase*)。它的子類🄵 *BufferedWriter*、*BufferedReader* 與 *BufferedRWPair* 分🄵緩衝可寫、可讀、可讀也可寫的的原

始二進位串流。類🄵*BufferedRandom* 則提供一個對可搜尋串流 (seekable stream) 的緩衝介面。另一個類🄵*BufferedIOBase* 的子類🄵*BytesIO*，是一個記憶體🄵位元組串流。

抽象基底類🄵*TextIOBase* 繼承*IOBase*。此類🄵處理文本位元組串流，🄵處理字串的編碼和解碼。類🄵*TextIOWrapper* 繼承自*TextIOBase*，這是個對緩衝原始串流 (*BufferedIOBase*) 的緩衝文本介面。最後，*StringIO* 是個文字記憶體🄵串流。

引數名稱不是規範的一部份，只有*open()* 的引數將作🄵關鍵字引數。

以下表格總結了*io* 模組提供的抽象基底類🄵 (ABC)：

抽象基底類🄵 (ABC)	繼承	Stub 方法	Mixin 方法與屬性
IOBase		`fileno`、`seek` 和 `truncate`	`close`、`closed`、`__enter__`、`__exit__`、`flush`、`isatty`、`__iter__`、`__next__`、`readable`、`readline`、`readlines`、`seekable`、`tell`、`writable` 與 `writelines`
RawIOBase	*IOBase*	`readinto` 和 `write`	繼承自*IOBase* 的方法，`read` 與 `readall`
BufferedIOBase	*IOBase*	`detach`、`read`、`read1` 和 `write`	繼承自*IOBase* 的方法，`readinto` 與 `readinto1`
TextIOBase	*IOBase*	`detach`、`read`、`readline` 和 `write`	繼承自*IOBase* 的方法，`encoding`、`errors` 與 `newlines`

I/O 基礎類🄵

class io.**IOBase**

所有 I/O 類🄵的抽象基礎類🄵。

🄵許多方法提供了空的抽象實作，衍生類🄵可以選擇性地覆寫這些方法；預設的實作代表一個無法讀取、寫入或搜尋的檔案。

即使*IOBase* 因🄵實作的簽名差🄵巨大而🄵有宣告 `read()` 或 `write()` 方法，實作與用🄵端應把這些方法視🄵介面的一部份。此外，當呼叫不被它們支援的操作時，可能會引發*ValueError*（或*UnsupportedOperation*）例外。

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. Text I/O classes work with *str* data.

請注意，在一個已經關閉的串流上呼叫任何方法（即使只是查詢）都是未定義的。在這種情🄵下，實作可能會引發*ValueError* 例外。

IOBase (and its subclasses) supports the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See *readline()* below.

IOBase 也是個情境管理器，因此支援 `with` 陳述式。在這個例子中，*file* 會在 `with` 陳述式執行完畢後關閉——即使發生了🄵常。

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase 提供這些資料屬性與方法：

close()

清除🄵關閉這個串流。若檔案已經關閉，則此方法🄵有作用。一旦檔案被關閉，任何對檔案的操作（例如讀取或寫入）將引發*ValueError* 🄵常。

🄵了方便起見，允許多次呼叫這個方法；然而，只有第一次呼叫會有效果。

closed

如果串流已關閉，則🄵 `True`。

fileno()

如果串流存在，則回傳其底層的檔案描述器（一個整數）。如果 IO 物件不使用檔案描述器，則會引發一個*OSError* 例外。

flush()

如果適用，清空串流的寫入緩衝區。對於唯讀和非阻塞串流，此操作不會執行任何操作。

isatty()

如果串流是互動式的（即連接到終端機/tty 設備），則回傳 `True`。

readable()

如果串流可以被讀取，則回傳 `True`。如果是 `False`，`read()` 將會引發*OSError* 例外。

readline(*size=-1*, */*)

從串流讀取🄵回傳一行。如果指定了 *size*，則最多讀取 *size* 個位元組。

對於二進位檔案，行結束符總是 `b'\n'`；對於文字檔案，可以使用*open()* 函式的 *newline* 引數來選擇識🄵的行結束符號。

readlines(*hint=-1*, */*)

從串流讀取🄵回傳一個含有一或多行的 list。可以指定 *hint* 來控制讀取的行數：如果到目前🄵止所有行的總大小（以位元組/字元計）超過 *hint*，則不會再讀取更多行。

hint 值🄵 `0` 或更小，以及 `None`，都被視🄵🄵有提供 hint。

請注意，已經可以使用 `for line in file: ...` 在檔案物件上進行🄵代，而不一定需要呼叫 `file.readlines()`。

seek(*offset*, *whence=os.SEEK_SET*, */*)

將串流位置改變到給定的位元組 *offset*，此位置是相對於由 *whence* 指示的位置解釋的，🄵回傳新的🄵對位置。*whence* 的值可🄵：

- *os.SEEK_SET* 或 `0` -- 串流的起點（預設值）；*offset* 應🄵零或正數
- *os.SEEK_CUR* 或 `1` -- 目前串流位置；*offset* 可以是負數
- *os.SEEK_END* 或 `2` -- 串流的結尾；*offset* 通常是負數

在 3.1 版被加入: `SEEK_*` 常數。

在 3.3 版被加入: 某些作業系統可以支援額外的值，例如*os.SEEK_HOLE* 或*os.SEEK_DATA*。檔案的合法值取🄵於它是以文字模式還是二進位模式開🄵。

seekable()

如果串流支援隨機存取，則回傳 `True`。如果是 `False`，則*seek()*、*tell()* 和*truncate()* 會引發*OSError*。

tell()

回傳目前串流的位置。

truncate(*size=None*, */*)

將串流的大小調整🄵指定的 *size* 位元組（如果🄵有指定 *size*，則調整🄵目前位置）。目前串流位置不會改變。這種調整可以擴展或縮🄵目前檔案大小。在擴展的情🄵下，新檔案區域的🄵容取🄵於平台（在大多數系統上，額外的位元組會被填充🄵零）。回傳新的檔案大小。

在 3.5 版的變更: Windows 現在在擴展時會對檔案進行零填充 (zero-fill)。

writable()

如果串流支援寫入，則回傳 `True`。如果是 `False`，`write()` 和*truncate()* 將會引發*OSError*。

writelines(*lines*, /)

將一個包含每一行的 list 寫入串流。這不會新增行分隔符號，因此通常提供的每一行末尾都有一個行分隔符號。

__del__()

🄵物件銷🄵做准備。*IOBase* 提供了這個方法的預設實作，該實作會呼叫實例的*close()* 方法。

class io.**RawIOBase**

原始二進位串流的基底類🄵。它繼承自*IOBase*。

原始二進位串流通常提供對底層作業系統設備或 API 的低階存取，🄵不嘗試將其封裝在高階基元 (primitive) 中（這項功能在緩衝二進位串流和文字串流中的更高階層級完成，後面的頁面會有描述）。

RawIOBase 除了*IOBase* 的方法外，還提供以下這些方法：

read(*size=-1*, /)

從物件中讀取最多 *size* 個位元組🄵回傳。方便起見，如果 *size* 未指定或🄵 -1，則回傳直到檔案結尾 (EOF) 的所有位元組。否則，只會進行一次系統呼叫。如果作業系統呼叫回傳的位元組少於 *size*，則可能回傳少於 *size* 的位元組。

如果回傳了 0 位元組，且 *size* 不是 0，這表示檔案結尾 (end of file)。如果物件處於非阻塞模式且🄵有可用的位元組，則回傳 None。

預設的實作會遵守*readall()* 和*readinto()* 的實作。

readall()

讀取🄵回傳串流中直到檔案結尾的所有位元組，必要時使用多次對串流的呼叫。

readinto(*b*, /)

將位元組讀入一個預先分配的、可寫的*bytes-like object*（類位元組物件）*b* 中，🄵回傳讀取的位元組數量。例如，*b* 可能是一個*bytearray*。如果物件處於非阻塞模式且🄵有可用的位元組，則回傳 None。

write(*b*, /)

將給定的*bytes-like object*（類位元組物件），*b*，寫入底層的原始串流，🄵回傳寫入的位元組大小。根據底層原始串流的具體情🄵，這可能少於 *b* 的位元組長度，尤其是當它處於非阻塞模式時。如果原始串流設置🄵非阻塞且無法立即寫入任何單一位元組，則回傳 None。呼叫者在此方法回傳後可以釋放或變更 *b*，因此實作應該只在方法呼叫期間存取 *b*。

class io.**BufferedIOBase**

支援某種緩衝的二進位串流的基底類🄵。它繼承自*IOBase*。

與*RawIOBase* 的主要差🄵在於，*read()*、*readinto()* 及*write()* 方法將分🄵嘗試讀取所請求的盡可能多的輸入，或發出所有提供的資料。

In addition, if the underlying raw stream is in non-blocking mode, when the system returns would block *write()* will raise *BlockingIOError* with *BlockingIOError.characters_written* and *read()* will return data read so far or None if no data is available.

此外，*read()* 方法不存在一個遵從*readinto()* 的預設實作。

一個典型的*BufferedIOBase* 實作不應該繼承自一個*RawIOBase* 的實作，而是應該改用包裝的方式，像*BufferedWriter* 和*BufferedReader* 那樣的作法。

BufferedIOBase 除了提供或覆寫來自*IOBase* 的資料屬性和方法以外，還包含了這些：

raw

底層的原始串流（一個*RawIOBase* 實例），*BufferedIOBase* 處理的對象。這不是*BufferedIOBase* API 的一部分，且在某些實作可能不存在。

detach()

將底層的原始串流從緩衝區中分離出來，🄵回傳它。

在原始串流被分離後，緩衝區處於一個不可用的狀態。

某些緩衝區，如*BytesIO*，🄵有單一原始串流的概念可從此方法回傳。它們會引發*UnsupportedOperation*。

在 3.1 版被加入.

read(*size=-1*, */*)

讀取🄵回傳最多 *size* 個位元組。如果引數被省略、🄵 None 或🄵負值，將讀取盡可能最多的位元組。

Fewer bytes may be returned than requested. An empty *bytes* object is returned if the stream is already at EOF. More than one read may be made and calls may be retried if specific errors are encountered, see *os.read()* and **PEP 475** for more details. Less than size bytes being returned does not imply that EOF is imminent.

When reading as much as possible the default implementation will use `raw.readall` if available (which should implement *RawIOBase.readall()*), otherwise will read in a loop until read returns `None`, an empty *bytes*, or a non-retryable error. For most streams this is to EOF, but for non-blocking streams more data may become available.

> **備🄵**
>
> When the underlying raw stream is non-blocking, implementations may either raise *BlockingIOError* or return `None` if no data is available. *io* implementations return `None`.

read1(*size=-1*, */*)

Read and return up to *size* bytes, calling *readinto()* which may retry if *EINTR* is encountered per **PEP 475**. If *size* is `-1` or not provided, the implementation will choose an arbitrary value for *size*.

> **備🄵**
>
> When the underlying raw stream is non-blocking, implementations may either raise *BlockingIOError* or return `None` if no data is available. *io* implementations return `None`.

readinto(*b*, */*)

讀取位元組到一個預先分配的、可寫的*bytes-like object* *b* 當中，🄵回傳讀取的位元組數量。例如，*b* 可能是一個*bytearray*。

類似於*read()*，除非後者是互動式的，否則可能會對底層原始串流發出多次讀取。

如果底層原始串流處於非阻塞模式，且目前🄵有可用資料，則會引發*BlockingIOError*。

readinto1(*b*, */*)

讀取位元組到一個預先分配的、可寫的*bytes-like object* *b* 中，最多呼叫一次底層原始串流的*read()*（或*readinto()*）方法。此方法回傳讀取的位元組數量。

如果底層原始串流處於非阻塞模式，且目前🄵有可用資料，則會引發*BlockingIOError*。

在 3.5 版被加入.

write(*b*, */*)

寫入給定的*bytes-like object*，*b*，🄵回傳寫入的位元組數量（總是等於 *b* 的長度，以位元組計，因🄵如果寫入失敗將會引發*OSError*）。根據實際的實作，這些位元組可能會立即寫入底層串流，或出於性能和延遲的緣故而被留在緩衝區當中。

當處於非阻塞模式時，如果需要將資料寫入原始串流，但它無法接受所有資料而不阻塞，則會引發*BlockingIOError*。

呼叫者可以在此方法回傳後釋放或變更 *b*，因此實作應該僅在方法呼叫期間存取 *b*。

原始檔案 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

一個代表包含位元組資料的 OS 層級檔案的原始二進制串流。它繼承自*RawIOBase*。

name 可以是兩種事物之一：

- 代表將要打開的檔案路徑的一個字元串或*bytes* 物件。在這種情🄵下，closefd 必須是 `True`（預設值），否則將引發錯誤。
- an integer representing the number of an existing OS-level file descriptor to which the resulting *FileIO* object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. *FileExistsError* will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The *read()* (when called with a positive argument), *readinto()* and *write()* methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener* must return an open file descriptor (passing *os.open* as *opener* results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

See the *open()* built-in function for examples on using the *opener* parameter.

在 3.3 版的變更: The *opener* parameter was added. The `'x'` mode was added.

在 3.4 版的變更: The file is now non-inheritable.

FileIO provides these data attributes in addition to those from *RawIOBase* and *IOBase*:

mode

The mode as given in the constructor.

name

The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.**BytesIO**(*initial_bytes=b''*)

A binary stream using an in-memory bytes buffer. It inherits from *BufferedIOBase*. The buffer is discarded when the *close()* method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

BytesIO provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

getbuffer()

Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

> **備🄵**
>
> As long as the view exists, the `BytesIO` object cannot be resized or closed.

在 3.2 版被加入.

getvalue()

Return `bytes` containing the entire contents of the buffer.

read1(*size=-1*, /)

In `BytesIO`, this is the same as `read()`.

在 3.7 版的變更: The *size* argument is now optional.

readinto1(*b*, /)

In `BytesIO`, this is the same as `readinto()`.

在 3.5 版被加入.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a readable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a `BufferedReader` for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, `DEFAULT_BUFFER_SIZE` is used.

`BufferedReader` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

peek(*size=0*, /)

Return bytes from the stream without advancing the position. The number of bytes returned may be less or more than requested. If the underlying raw stream is non-blocking and the operation would block, returns empty bytes.

read(*size=-1*, /)

In `BufferedReader` this is the same as `io.BufferedIOBase.read()`

read1(*size=-1*, /)

In `BufferedReader` this is the same as `io.BufferedIOBase.read1()`

在 3.7 版的變更: The *size* argument is now optional.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a writeable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying `RawIOBase` object under various conditions, including:

- when the buffer gets too small for all pending data;
- when `flush()` is called;
- when a `seek()` is requested (for `BufferedRandom` objects);
- when the `BufferedWriter` object is closed or destroyed.

The constructor creates a `BufferedWriter` for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

flush()

Force bytes held in the buffer into the raw stream. A *BlockingIOError* should be raised if the raw stream blocks.

write(*b*, /)

Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a *BlockingIOError* with *BlockingIOError.characters_written* set is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a seekable *RawIOBase* raw binary stream. It inherits from *BufferedReader* and *BufferedWriter*.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRandom is capable of anything *BufferedReader* or *BufferedWriter* can do. In addition, *seek()* and *tell()* are guaranteed to be implemented.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)

A buffered binary stream providing higher-level access to two non seekable *RawIOBase* raw binary streams---one readable, the other writeable. It inherits from *BufferedIOBase*.

reader and *writer* are *RawIOBase* objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRWPair implements all of *BufferedIOBase*'s methods except for *detach()*, which raises *UnsupportedOperation*.

> **⚠ 警告**
>
> *BufferedRWPair* does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use *BufferedRandom* instead.

文字 I/O

class io.**TextIOBase**

Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits from *IOBase*.

TextIOBase provides or overrides these data attributes and methods in addition to those from *IOBase*:

encoding

The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors

The error setting of the decoder or encoder.

newlines

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer

底層的二進位緩衝（一個*BufferedIOBase* 或*RawIOBase* 實例），🄵*TextIOBase* 處理的對象。這不是*TextIOBase* API 的一部分，且在某些實作可能不存在。

detach()

Separate the underlying binary buffer from the *TextIOBase* and return it.

After the underlying buffer has been detached, the *TextIOBase* is in an unusable state.

Some *TextIOBase* implementations, like *StringIO*, may not have the concept of an underlying buffer and calling this method will raise *UnsupportedOperation*.

在 3.1 版被加入.

read(*size=-1*, */*)

Read and return at most *size* characters from the stream as a single `str`. If *size* is negative or `None`, reads until EOF.

readline(*size=-1*, */*)

Read until newline or EOF and return a single `str`. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*, */*)

Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is `SEEK_SET`.

- `SEEK_SET` or `0`: seek from the start of the stream (the default); *offset* must either be a number returned by `TextIOBase.tell()`, or zero. Any other *offset* value produces undefined behaviour.
- `SEEK_CUR` or `1`: "seek" to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).
- `SEEK_END` or `2`: seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

在 3.1 版被加入: `SEEK_*` 常數。

tell()

Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*, */*)

Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

A buffered text stream providing higher-level access to a `BufferedIOBase` buffered binary stream. It inherits from `TextIOBase`.

encoding gives the name of the encoding that the stream will be decoded or encoded with. In *UTF-8 Mode*, this defaults to UTF-8. Otherwise, it defaults to `locale.getencoding()`. `encoding="locale"` can be used to specify the current locale's encoding explicitly. See 文字編碼 for more information.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass `'strict'` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. `'backslashreplace'` causes malformed data to be replaced by a backslashed escape sequence. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'namereplace'` (replace with `\N{...}` escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline controls how line endings are handled. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, if *newline* is `None`, *universal newlines* mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If *newline* is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If *newline* has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If *newline* is `''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *line_buffering* is `True`, *flush()* is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to *write()* are guaranteed not to be buffered: any data written on the *TextIOWrapper* object is immediately handled to its underlying binary *buffer*.

在 3.3 版的變更: The *write_through* argument has been added.

在 3.3 版的變更: The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporary the locale encoding using *locale.setlocale()*, use the current locale encoding instead of the user preferred encoding.

在 3.10 版的變更: The *encoding* argument now supports the `"locale"` dummy encoding name.

備🄵

When the underlying raw stream is non-blocking, a *BlockingIOError* may be raised if a read operation cannot be completed immediately.

TextIOWrapper provides these data attributes and methods in addition to those from *TextIOBase* and *IOBase*:

line_buffering

Whether line buffering is enabled.

write_through

Whether writes are passed immediately to the underlying binary buffer.

在 3.7 版被加入.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

在 3.7 版被加入.

在 3.11 版的變更: The method supports `encoding="locale"` option.

seek(*cookie*, *whence=os.SEEK_SET*, */*)

Set the stream position. Return the new stream position as an *int*.

Four operations are supported, given by the following argument combinations:

- `seek(0, SEEK_SET)`: Rewind to the start of the stream.
- `seek(cookie, SEEK_SET)`: Restore a previous position; *cookie* **must be** a number returned by *tell()*.
- `seek(0, SEEK_END)`: Fast-forward to the end of the stream.
- `seek(0, SEEK_CUR)`: Leave the current stream position unchanged.

Any other argument combinations are invalid, and may raise exceptions.

也參考

os.SEEK_SET, *os.SEEK_CUR*, and *os.SEEK_END*.

tell()

Return the stream position as an opaque number. The return value of `tell()` can be given as input to *seek()*, to restore a previous stream position.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

A text stream using an in-memory text buffer. It inherits from *TextIOBase*.

The text buffer is discarded when the *close()* method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by *write()*. The stream is positioned at the start of the buffer which emulates opening an existing file in a `w+` mode, making it ready for an immediate write from the beginning or for a write that would overwrite the initial value. To emulate opening a file in an `a+` mode ready for appending, use `f.seek(0, io.SEEK_END)` to reposition the stream at the end of the buffer.

The *newline* argument works like that of *TextIOWrapper*, except that when writing output to the stream, if *newline* is `None`, newlines are written as `\n` on all platforms.

StringIO provides this method in addition to those from *TextIOBase* and *IOBase*:

getvalue()

Return a *str* containing the entire contents of the buffer. Newlines are decoded as if by *read()*, although the stream position is not changed.

使用範例:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**

A helper codec that decodes newlines for *universal newlines* mode. It inherits from *codecs.IncrementalDecoder*.

16.2.5 Static Typing

The following protocols can be used for annotating function and method arguments for simple stream reading or writing operations. They are decorated with *@typing.runtime_checkable*.

class io.**Reader**[*T*]()

Generic protocol for reading from a file or other input stream. `T` will usually be *str* or *bytes*, but can be any type that is read from the stream.

在 3.14 版被加入.

read()

read(*size*, */*)

Read data from the input stream and return it. If *size* is specified, it should be an integer, and at most *size* items (bytes/characters) will be read.

例如:

```
def read_it(reader: Reader[str]):
    data = reader.read(11)
    assert isinstance(data, str)
```

class io.**Writer**[*T*]()

Generic protocol for writing to a file or other output stream. `T` will usually be *str* or *bytes*, but can be any type that can be written to the stream.

在 3.14 版被加入.

write(*data*, /)

Write *data* to the output stream and return the number of items (bytes/characters) written.

例如:

```
def write_binary(writer: Writer[bytes]):
    writer.write(b"Hello world!\n")
```

See *ABCs and Protocols for working with I/O* for other I/O related protocols and classes that can be used for static type checking.

16.2.6 Performance

This section discusses the performance of the provided concrete I/O implementations.

二進位 (Binary) I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

文字 I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, *tell()* and *seek()* are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as *read(2)* under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in *print()* function as well.

16.3 time --- 時間存取與轉🄵

這個模組提供了各種與時間相關的函式。若要查看相關功能，請參🄵*datetime* 和*calendar* 模組。

雖然這個模組隨時可用，但🄵非所有函式在所有平台上都可用。這個模組中定義的大多數函式都會呼叫 C 語言平台的函式庫中具有相同名稱的函式。由於這些函式的語義因平台而🄵，所以偶爾查🄵平台文件可能會有所幫助。

以下是對一些術語和慣例的🄵明。

- *epoch* 是起始的時間點，即 `time.gmtime(0)` 的回傳值。在所有平台上，它是 1970 年 1 月 1 日，00:00:00（UTC）。
- 術語 *seconds since the epoch*（紀元秒數）是指從 epoch（紀元）開始經過的總秒數，通常不包括 leap seconds。在所有符合 POSIX 標準的平台上，leap seconds （閏秒）都不計入這個總數。
- 這個模組中的函式可能無法處理*epoch* 之前或🄵遠未來的日期和時間。未來的臨界點由 C 函式庫🄵定；對於 32 位元系統來🄵通常是在 2038 年。
- 函式*strptime()* 在給定 `%y` 格式碼時可以剖析 (parse) 兩位數的年份。當剖析兩位數的年份時，它們會根據 POSIX 和 ISO C 標準進行轉🄵：69--99 的值對映到 1969--1999，0--68 的值對映到 2000--2068。
- UTC is Coordinated Universal Time and superseded Greenwich Mean Time or GMT as the basis of international timekeeping. The acronym UTC is not a mistake but conforms to an earlier, language-agnostic naming scheme for time standards such as UT0, UT1, and UT2.
- DST 是 Daylight Saving Time（日光節約時間），一年中的某些時段（通常）將會時區調整一小時。DST 的規則是根據當地法律🄵定的，且可能每年不同。C 函式庫有一個包含當地規則的表（通常會🄵了靈活性而從系統文件中讀取），在這方面是唯一的真正依據。
- 各種即時 (real-time) 函式的精確度可能低於其值或引數所表示的單位所建議的精確度。例如，在大多數 Unix 系統上，時鐘每秒只「跳」50 次或 100 次。
- 另一方面，*time()* 和*sleep()* 的精確度比它們的在 Unix 的等效函式更高：時間以浮點數表示，*time()* 回傳最精確的可用時間（如果可以會使用 Unix 的 `gettimeofday()`），而*sleep()* 可以接受帶有非零分數的時間（如果可以會使用 Unix 的 `select()` 來實作）。
- 由*gmtime()*、*localtime()* 和*strptime()* 回傳，🄵由*asctime()*、*mktime()* 和*strftime()* 接受的時間值，是一個 9 個整數的序列。*gmtime()*、*localtime()* 和*strptime()* 的回傳值也🄵各個欄位提供屬性名稱。

  關於這些物件的🄵述請見*struct_time*。

  在 3.3 版的變更：當平台支援對應的 `struct tm` 成員時，*struct_time* 型🄵被擴展以提供*tm_gmtoff* 和*tm_zone* 屬性。

  在 3.6 版的變更：*struct_time* 的屬性*tm_gmtoff* 和*tm_zone* 現在在所有平台上都可用。
- 使用以下函式在時間表示之間進行轉🄵：

轉🄵來源	轉🄵目標	使用
紀元秒數	世界協調時間的*struct_time*	*gmtime()*
紀元秒數	本地時間的*struct_time*	*localtime()*
世界協調時間的*struct_time*	紀元秒數	*calendar.timegm()*
本地時間的*struct_time*	紀元秒數	*mktime()*

16.3.1 函式

`time.`**`asctime`**`(`[*t*]`)`

將由*gmtime()* 或*localtime()* 回傳的元組或*struct_time* 表示的時間轉🄵🄵以下格式的字串：`'Sun Jun 20 23:21:05 1993'`。日期欄位🄵兩個字元長，如果日期是個位數，則用空格填充，例如：`'Wed Jun 9 04:26:40 1993'`。

如果🄵有提供 *t*，則使用由`localtime()` 回傳的目前時間。`asctime()` 不使用區域資訊。

> **備🄵**
>
> 與同名的 C 函式不同，`asctime()` 不會添加結尾的🄵行字元。

time.**pthread_getcpuclockid**(*thread_id*)

🄵指定的 *thread_id* 回傳執行緒專用 CPU-time 時鐘的 *clk_id*。

使用`threading.get_ident()` 或`threading.Thread` 物件的`ident` 屬性來取得適用於 *thread_id* 的值。

> **警告**
>
> 傳遞無效或過期的 *thread_id* 可能會導致未定義的行🄵，例如分段錯誤 (segmentation fault)。

可用性: Unix

若需更多資訊，請參🄵 `pthread_getcpuclockid(3)` 的🄵明文件。

在 3.7 版被加入.

time.**clock_getres**(*clk_id*)

回傳指定時鐘 *clk_id* 的解析度（精確度）。有關 *clk_id* 可接受的值的串列，請參🄵時鐘 *ID* 常數。

可用性: Unix.

在 3.3 版被加入.

time.**clock_gettime**(*clk_id*) → *float*

回傳指定時鐘 *clk_id* 的時間。有關 *clk_id* 可接受的值的串列，請參🄵時鐘 *ID* 常數。

使用`clock_gettime_ns()` 以避免`float` 型🄵造成的精確度損失。

可用性: Unix.

在 3.3 版被加入.

time.**clock_gettime_ns**(*clk_id*) → *int*

類似於`clock_gettime()`，但回傳以奈秒 (nanoseconds) 🄵單位的時間。

可用性: Unix.

在 3.7 版被加入.

time.**clock_settime**(*clk_id*, *time:* float)

設定指定時鐘 *clk_id* 的時間。目前，`CLOCK_REALTIME` 是 *clk_id* 唯一可以接受的值。

使用`clock_settime_ns()` 以避免`float` 型🄵造成的精確度損失。

可用性: Unix, not Android, not iOS.

在 3.3 版被加入.

time.**clock_settime_ns**(*clk_id*, *time:* int)

類似於`clock_settime()`，但設定以奈秒🄵單位的時間。

可用性: Unix, not Android, not iOS.

在 3.7 版被加入.

time.**ctime**([*secs*])

將自*epoch*起以秒表示的時間轉🄵🄴表示當地時間且符合以下格式的字串：`'Sun Jun 20 23:21:05 1993'`。日期欄位🄵兩個字元長，如果日期是個位數，則用空格填充，例如：`'Wed Jun 9 04:26:40 1993'`。

如果未提供 *secs* 或其🄵*None*，則使用由*time()* 回傳的目前時間。`ctime(secs)` 等同於 `asctime(localtime(secs))`。*ctime()* 不使用區域資訊。

time.**get_clock_info**(*name*)

取得指定時鐘的資訊作🄵命名空間物件。支援的時鐘名稱及讀取他們的值的對應函式如下：

- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'thread_time'`: *time.thread_time()*
- `'time'`: *time.time()*

其結果具有以下屬性：

- *adjustable*: 如果時鐘可以自動（例如，透過 NTP 常駐程式）或由系統管理員手動更改，則🄵 `True`，否則🄵 `False`
- *implementation*: 用於取得時鐘的值的底層 C 函式名稱。有關可能的值，請參🄵*時鐘 ID 常數*。
- *monotonic*: 如果時鐘不能倒轉，則🄵 `True`，否則🄵 `False`
- *resolution*: 以秒 (*float*) 🄵單位的時鐘的解析度

在 3.3 版被加入.

time.**gmtime**([*secs*])

將自*epoch* 起以秒表示的時間轉🄵🄴 UTC 中的*struct_time*，其中 dst 旗標始終🄵零。如果未提供 *secs* 或其🄵*None*，則使用由*time()* 回傳的目前時間。忽略秒的分數部分。關於*struct_time* 物件的描述，請參🄵上文。此函式的反運算請參🄵*calendar.timegm()*。

time.**localtime**([*secs*])

類似於*gmtime()*，但轉🄵🄴當地時間。如果未提供 *secs* 或其🄵*None*，則使用由*time()* 回傳的目前時間。當 DST 適用於給定時間時，dst 旗標會被設定🄵 `1`。

如果時間戳超出 C 平台的 `localtime()` 或 `gmtime()` 函式支援的範圍，*localtime()* 可能會引發*OverflowError*；在 `localtime()` 或 `gmtime()` 失敗時，會引發*OSError*。通常會把年份限制在 1970 年到 2038 年之間。

time.**mktime**(*t*)

這是*localtime()* 的反函式。其引數是表示當地時間（不是 UTC）的*struct_time* 或完整的 9 元組（因🄵需要 dst 旗標；如果 dst 🄵未知，則使用 `-1` 作🄵 dst 旗標）。它回傳一個浮點數，以與*time()* 相容。如果輸入值不能表示🄵有效時間，將引發*OverflowError* 或*ValueError*（取🄵於無效值是被 Python 還是底層 C 函式庫捕獲）。它能生成時間的最早日期根據平台而有所不同。

time.**monotonic**() → *float*

回傳單調時鐘（monotonic clock，即不能倒轉的時鐘）的值（以帶有小數的秒數表示）。該時鐘不受系統時鐘更新的影響。回傳值的參考點🄵有定義，因此只有兩次呼叫結果之間的差🄵才是有效的。

時鐘：

- 在 Windows 上，呼叫 `QueryPerformanceCounter()` 和 `QueryPerformanceFrequency()`。
- 在 macOS 上，呼叫 `mach_absolute_time()` 和 `mach_timebase_info()`。
- 在 HP-UX 上，呼叫 `gethrtime()`。
- 如果可以的話，呼叫 `clock_gettime(CLOCK_HIGHRES)`。
- 否則，呼叫 `clock_gettime(CLOCK_MONOTONIC)`。

使用*monotonic_ns()* 以避免*float* 型🄵造成的精確度損失。

在 3.3 版被加入.

在 3.5 版的變更: 此函式現在始終可用且時鐘對於所有行程都會是相同。

在 3.10 版的變更: 在 macOS 上，時鐘對於所有行程都會是相同。

time.**monotonic_ns**() → *int*

類似於*monotonic()*，但回傳以奈秒🄵單位的時間。

在 3.7 版被加入.

time.**perf_counter**() → *float*

回傳性能計數器的值（以帶有小數的秒數表示），即具有最高可用解析度來測量短時間間隔的時鐘。它包括睡眠時經過的時間，且時鐘對於所有行程都會是相同。回傳值的參考點🄵有定義，因此只有兩次呼叫結果之間的差🄵才是有效的。

在 CPython 上，使用與*time.monotonic()* 相同的時鐘，且其🄵單調時鐘（即不能倒轉的時鐘）。

使用*perf_counter_ns()* 以避免*float* 型🄵造成的精確度損失。

在 3.3 版被加入.

在 3.10 版的變更: 在 Windows 上，時鐘對於所有行程都會是相同。

在 3.13 版的變更: 使用與*time.monotonic()* 相同的時鐘。

time.**perf_counter_ns**() → *int*

類似於*perf_counter()*，但回傳以奈秒🄵單位的時間。

在 3.7 版被加入.

time.**process_time**() → *float*

回傳目前行程的系統和用🄵 CPU 時間之和（以帶有小數的秒數表示）。它不包括睡眠時經過的時間。根據定義，它涵蓋整個行程。回傳值的參考點🄵有定義，因此只有兩次呼叫結果之間的差🄵才是有效的。

使用*process_time_ns()* 以避免*float* 型🄵造成的精確度損失。

在 3.3 版被加入.

time.**process_time_ns**() → *int*

類似於*process_time()*，但回傳以奈秒🄵單位的時間。

在 3.7 版被加入.

time.**sleep**(*secs*)

在一個給定的秒數🄵暫停呼叫執行緒 (calling thread) 的執行。引數可以是浮點數，以表示更精確的睡眠時間。

如果睡眠被訊號中斷且訊號處理器未引發例外，則睡眠將以重新計算過的逾時 (timeout) 重新開始。

由於系統中其他活動的調度，暫停時間可能會比請求的時間長任意的量。

Windows 實作

在 Windows 上，如果 *secs* 🄵零，則執行緒將其剩余的時間片段讓給任何准備運行的其他執行緒。如果🄵有其他執行緒准備運行，該函式將立即回傳，而執行緒會繼續執行。在 Windows 8.1 及更新的版本中，此實作使用高解析度計時器，其解析度🄵 100 奈秒。如果 *secs* 🄵零，則使用 `Sleep(0)`。

Unix 實作

- 如果可以，使用 `clock_nanosleep()`（解析度：1 奈秒）；
- 或者使用 `nanosleep()`（解析度：1 奈秒）；
- 或使用 `select()`（解析度：1 微秒）。

> **備🄵**
>
> To emulate a "no-op", use `pass` instead of `time.sleep(0)`.
>
> To voluntarily relinquish the CPU, specify a real-time *scheduling policy* and use *os.sched_yield()* instead.

引發一個帶有引數 `secs` 的稽核事件 *(auditing event)* `time.sleep`。

在 3.5 版的變更: 即使睡眠被訊號中斷，此函式現在至少還是會睡眠 *secs*，除非訊號處理器引發例外 (理由請參🄵 **PEP 475**)。

在 3.11 版的變更: 在 Unix 上，如果可以的話現在會使用 `clock_nanosleep()` 和 `nanosleep()` 函式。在 Windows 上，現在使用可等待的計時器。

在 3.13 版的變更: 引發一個稽核事件。

`time.`**`strftime`**(*format*[, *t*])

將由 *gmtime()* 或 *localtime()* 回傳代表時間的一個元組或 *struct_time* 轉🄵🄵由 *format* 引數指定的字串。如果未提供 *t*，則使用由 *localtime()* 回傳的目前時間。*format* 必須是一個字串。如果 *t* 中的任何欄位超出允許範圍，將會引發 *ValueError*。

0 在時間元組中的任何位置都是合法引數；如果元組中出現常見的錯誤，該值將被🄵制更改🄵正確的值。

以下指令可以嵌入在 *format* 字串中。它們顯示時不帶可選的欄位寬度和精度規範，🄵在 *strftime()* 的結果中被標示的字元替🄵：

指令	意義	F解
%a	區域設定的F間日 (weekday) 縮寫名稱。	
%A	區域設定的完整F間日名稱。	
%b	區域設定的縮寫月份名稱。	
%B	區域設定的完整月份名稱。	
%c	區域設定的合適的日期和時間的表示法。	
%d	月份中的日期，表示F十進位數 [01,31]。	
%f	**微秒，表示F十進位數** [000000,999999]。	(1)
%H	小時（24 小時制），表示F十進位數 [00,23]。	
%I	小時（12 小時制），表示F十進位數 [01,12]。	
%j	一年中的第幾天，表示F十進位數 [001,366]。	
%m	月份，表示F十進位數 [01,12]。	
%M	分鐘，表示F十進位數 [00,59]。	
%p	區域設定中相當於 AM 或 PM 的表示。	(2)
%S	秒，表示F十進位數 [00,61]。	(3)
%U	一年中的F數（星期天作F一F的第一天），表示F十進位數 [00,53]。新的一年中，在第一個星期天之前的所有日子都被認定F第 0 F。	(4)
%u	一F中的日期（周一F 1；F日F 7），表示F十進位數 [1,7]。	
%w	F間日，表示F十進位數 [0（星期天）,6]。	
%W	一年中的F數（星期一作F一F的第一天），表示F十進位數 [00,53]。新的一年中，在第一個星期一之前的所有日子都被認定F第 0 F。	(4)
%x	區域設定的合適的日期表示法。	
%X	區域設定的合適的時間表示法。	
%y	去掉世紀的年份，表示F十進位數 [00,99]。	
%Y	帶世紀的年份，表示F十進位數。	
%z	時區偏移量，表示與 UTC/GMT 的正或負時間差，形式F +HHMM 或 -HHMM，其中 H 代表十進位的小時數碼 (digits)，M 代表十進位的分鐘數碼 [-23:59, +23:59]。[1]	
%Z	時區名稱（如果不存在時區，則無字元）。已被F用。Page 754, 1	
%G	ISO 8601 年（類似於 %Y，但遵循 ISO 8601 日F年的規則）。	
%V	年份從包含該日F年第一個星期四的那一F開始。	
%V	ISO 8601 F數（以十進位數表示 [01,53]）。年份的第一F是包含該年第一個星期四的那一	

🄵解：

(1) `%f` 格式的指令僅適用於*strptime()*，不適用於*strftime()*。然而，在*datetime.datetime.strptime()* 和*datetime.datetime.strftime()* 其中的 `%f` 格式的指令適用於微秒。

(2) 當與*strptime()* 函式一起使用時，`%p` 指令僅在使用 `%I` 指令剖析小時時影響輸出小時的欄位。

(3) 範圍確實是從 `0` 到 `61`；數值 `60` 在表示 leap seconds 的時間戳中是有效的，而數值 `61` 是出於歷史因素而被支援。

(4) 當與*strptime()* 函式一起使用時，`%U` 和 `%W` 僅在指定🄵間的某天和年份時用於計算中。

以下是一個範例，其🄵一種與 **RFC 2822** 網際網路電子郵件標準中指定的日期格式兼容的格式。[1]：

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

某些平台可能支援額外的指令，但只有這🄵列出的指令具有 ANSI C 標準化的意義。要查看你的平台上支援的完整格式碼集，請參🄵 *strftime(3)* 文件。

在某些平台上，可選的欄位寬度和精度規範可以以此順序緊跟在指令初始的 `'%'` 之後；這也是不可🄵 (portable) 的。欄位寬度通常🄵 2，除了 `%j` 🄵 3。

`time.`**`strptime`**`(`*string*`[, `*format*`])`

根據格式剖析表示時間的字串。回傳值是*struct_time*，如同由*gmtime()* 或*localtime()* 回傳的一樣。

format 參數使用與*strftime()* 相同的指令；預設🄵 `"%a %b %d %H:%M:%S %Y"`，與*ctime()* 回傳的格式匹配。如果 *string* 無法根據 *format* 解析，或剖析後有多余的資料，將引發*ValueError*。當無法推斷更精確的值時，用來填充任何缺失資料的預設值🄵 `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`。*string* 和 *format* 都必須是字串。

例如：

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

對 `%Z` 指令的支援基於 `tzname` 中包含的值以及 `daylight` 是否🄵 true。因此，除了識🄵始終已知的 UTC 和 GMT（且被考慮🄵非日光節約時區）外，這是特定於平台的。

僅支援文件中指定的指令。由於 `strftime()` 是根據每個平台實作的，有時它可以提供比列出的還要更多的指令。但是 `strptime()` 與任何平台無關，因此不一定支援所有未記載🄵支援的指令。

`class` `time.`**`struct_time`**

由*gmtime()*、*localtime()* 和*strptime()* 回傳的時間值序列的型🄵。它是一個具有*named tuple* 介面的物件：值可以通過索引和屬性名稱存取。包含以下值：

[1] 現在 `%Z` 已被🄵用，`%z` 藉由擴展🄵偏好的小時/分鐘偏移而逃離了被🄵用的命運，但其🄵不是被所有 ANSI C 函式庫支援。此外，嚴格按照 1982 年的 **RFC 822** 標準，應該使用兩位數年份（`%y` 而不是 `%Y`），但實作中在 2000 年之前就已經轉向使用四位數年份。之後，**RFC 822** 被淘汰，四位數年份首先由 **RFC 1123** 推薦，然後由 **RFC 2822** 正式批准。

索引	屬性	值
0	**tm_year**	（例如 1993）
1	**tm_mon**	範圍 [1, 12]
2	**tm_mday**	範圍 [1, 31]
3	**tm_hour**	範圍 [0, 23]
4	**tm_min**	範圍 [0, 59]
5	**tm_sec**	範圍 [0, 61]；參見*strftime()* 中的🄵釋 *(2)*
6	**tm_wday**	範圍 [0, 6]；星期一是 0
7	**tm_yday**	範圍 [1, 366]
8	**tm_isdst**	0、1 或 -1；見下文
N/A	**tm_zone**	時區名稱的縮寫
N/A	**tm_gmtoff**	UTC 向東的偏移量（以秒🄴單位）

請注意，與 C 結構不同，月份值的範圍是 [1, 12]，而不是 [0, 11]。

在呼叫*mktime()* 時，當日光節約時間生效的時候，*tm_isdst* 可以設定🄴 1，不生效時設定🄵 0。值 -1 表示未知是否生效，通常結果會填入正確的狀態。

當一個長度不正確的元組被傳遞給預期得到*struct_time* 的函式時，或者其中有元素型🄴錯誤時，將引發*TypeError*。

time.**time**() → *float*

回傳自*epoch* 起的時間（秒）至今的浮點數。對 leap seconds 的處理是與平台有關的。在 Windows 和大多數 Unix 系統上，閏秒不計入自*epoch* 起的秒數中。這通常被稱🄵 Unix 時間。

請注意，即使時間始終作🄵浮點數回傳，但🄵非所有系統都提供比 1 秒還更精確的時間。雖然此函式通常回傳非遞🄵的值，但如果在兩次呼叫之間系統時鐘被回調，則它可能回傳比之前呼叫更小的值。

由*time()* 回傳的數字可以通過傳遞給*gmtime()* 函式轉🄵🄵 UTC 🄵更常見的時間格式（即年、月、日、小時等）或通過傳遞給*localtime()* 函式轉🄵🄵當地時間。在這兩種情🄵下都會回傳一個*struct_time* 物件，從中可以作🄵屬性存取日🄵日期的組成部分。

時鐘：

- 在 Windows 上，呼叫 `GetSystemTimePreciseAsFileTime()`。
- 如果可以的話，呼叫 `clock_gettime(CLOCK_REALTIME)`。
- 否則，呼叫 `gettimeofday()`。

使用*time_ns()* 以避免*float* 型🄵造成的精確度損失。

在 3.13 版的變更: 在 Windows 上，呼叫 `GetSystemTimePreciseAsFileTime()` 而不是 `GetSystemTimeAsFileTime()`。

`time.`**`time_ns`**`()` → *int*

類似於*time()*，但回傳自*epoch* 起的以奈秒🄵單位的整數。

在 3.7 版被加入.

`time.`**`thread_time`**`()` → *float*

回傳目前執行緒的系統和用🄵 CPU 時間之和（以帶有小數的秒數表示）。它不包括睡眠期間經過的時間。根據定義，這是執行緒特定 (thread-specific) 的。回傳值的參照點未定義，因此只有同一執行緒中兩次呼叫結果之間的差🄵才是有效的。

使用*thread_time_ns()* 以避免*float* 型🄵造成的精確度損失。

可用性: Linux, Unix, Windows.

有支援 `CLOCK_THREAD_CPUTIME_ID` 的 Unix 系統。

在 3.7 版被加入.

`time.`**`thread_time_ns`**`()` → *int*

類似於*thread_time()*，但回傳以奈秒🄵單位的時間。

在 3.7 版被加入.

`time.`**`tzset`**`()`

重置函式庫常式 (routine) 使用的時間轉🄵規則。環境變數 `TZ` 指定了這一過程的實施方式。它還會設定變數 `tzname`（來自 `TZ` 環境變數）、`timezone`（非日光節約時間的 UTC 以西的時間偏移，單位🄵秒）、`altzone`（日光節約時間的 UTC 以西的時間偏移，單位🄵秒）和 `daylight`（如果該時區🄵有日光節約時間規則，則設定🄵 0；如果在過去、現在或未來的某個時間有日光節約時間規則，則設置🄵非零的值）。

可用性: Unix.

> **備🄵**
>
> 雖然在許多情🄵下，更改 `TZ` 環境變數可能會在🄵有呼叫*tzset()* 的情🄵下影響*localtime()* 等函式的輸出，但是這種行🄵是不該被依賴的。
>
> `TZ` 環境變數不應包含空格字元。

`TZ` 環境變數的標準格式🄵（🄵了清楚表達，中間增加了空格字元）:

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

其中各個組成部分🄵:

`std` 和 `dst`
三個或更多字母與數字 (alphanumerics) 組成的時區縮寫。這些縮寫會被傳播到 time.tzname 中。

`offset`
偏移量的格式🄵: `±` `hh[:mm[:ss]]`。這表示🄵達到 UTC 而增加到當地時間的值。如果以'-' 開頭，則表示該時區位於本初子午🄵以東；否則，位於其西。如果 dst 之後🄵有偏移量，則假定日光時間比標準時間快一小時。

`start[/time], end[/time]`
表示何時切🄵至日光節約時間及何時切🄵回來。開始和結束日期的格式如以下其一:

`Jn`
儒略日 (Julian day) *n** (*1* <= **n* <= 365)。閏日不計算，因此在所有年份中，2 月 28 日是第 59 天，3 月 1 日是第 60 天。

n
從 0 開始的儒略日 (0 <= *n* <= 365)。閏日會計算，因此可以適用至 2 月 29 日。

Mm.n.d
一年中第 *m* 月的第 *n* F的第 *d* 天（0 <= *d* <= 6，1 <= *n* <= 5，1 <= *m* <= 12，其中 *n* F 5 表示「該月的最後一個第 *d* 天」，這可能出現在第四或第五F)。第 1 F是 *d* 天首次出現的那一F。第零天F星期天。

`time` 的格式與 `offset` 相同，但不允許出現前導符號（'-' 或'+'）。如果未指定時間，則預設F 02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在許多 Unix 系統（包括 *BSD、Linux、Solaris 和 Darwin）上，使用系統的 zoneinfo (*tzfile(5)*) 資料庫來指定時區規則會更加方便。要這樣做，請將 `TZ` 環境變數設定F所需時區資料檔案的路徑，相對於系統'zoneinfo' 時區資料庫的根目F，通常位於 `/usr/share/zoneinfo`。例如，`'US/Eastern'`、`'Australia/Melbourne'`、`'Egypt'` 或 `'Europe/Amsterdam'`。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 時鐘 ID 常數

這些常數用作*clock_getres()* 和*clock_gettime()* 的參數。

`time.`**CLOCK_BOOTTIME**
與*CLOCK_MONOTONIC* 基本相同，不同之處在於它還包括系統暫停的任何時間。

這允許應用程式取得一個能F感知暫停的單調時鐘，而無需處理*CLOCK_REALTIME* 的F雜情F，後者在使用 `settimeofday()` 或類似函式更改時間時可能會出現不連續的情F。

可用性: Linux >= 2.6.39.

在 3.7 版被加入.

`time.`**CLOCK_HIGHRES**
Solaris 作業系統具有 `CLOCK_HIGHRES` 計時器，它嘗試使用最佳的硬體資源，F可能提供接近奈秒的解析度。`CLOCK_HIGHRES` 是不可調整且高解析度的時鐘。

可用性: Solaris.

在 3.3 版被加入.

`time.`**CLOCK_MONOTONIC**
該時鐘無法被設定，其表示自某個未指定起點以來的單調時間。

可用性: Unix.

在 3.3 版被加入.

time.**CLOCK_MONOTONIC_RAW**

類似於*CLOCK_MONOTONIC*，但提供對基於硬體的原始時間的存取，此時間不受 NTP 調整的影響。

可用性: Linux >= 2.6.28, macOS >= 10.12.

在 3.3 版被加入.

time.**CLOCK_MONOTONIC_RAW_APPROX**

類似於*CLOCK_MONOTONIC_RAW*，但讀取的是系統在情境切🄵時快取的值，因此精准度較低。

可用性: macOS >= 10.12.

在 3.13 版被加入.

time.**CLOCK_PROCESS_CPUTIME_ID**

來自 CPU 的高解析度每個行程的計時器。

可用性: Unix.

在 3.3 版被加入.

time.**CLOCK_PROF**

來自 CPU 的高解析度每個行程的計時器。

可用性: FreeBSD, NetBSD >= 7, OpenBSD.

在 3.7 版被加入.

time.**CLOCK_TAI**

國際原子時間

系統必須擁有目前的閏秒表才能給出正確答案。PTP 或 NTP 軟體可以維護閏秒表。

可用性: Linux.

在 3.9 版被加入.

time.**CLOCK_THREAD_CPUTIME_ID**

執行緒相關的 CPU 時間時鐘。

可用性: Unix.

在 3.3 版被加入.

time.**CLOCK_UPTIME**

表示系統運作且無暫停的🄵對時間，提供包括🄵對時間 (absolute) 和時間區間 (interval) 的精確的正常上🄵時間 (uptime) 測量。

可用性: FreeBSD, OpenBSD >= 5.5.

在 3.7 版被加入.

time.**CLOCK_UPTIME_RAW**

單調增量的時鐘，從某個任意點開始計時，不受頻率或時間調整影響，🄵且在系統休眠時不增量。

可用性: macOS >= 10.12.

在 3.8 版被加入.

time.**CLOCK_UPTIME_RAW_APPROX**

類似於*CLOCK_UPTIME_RAW*，但該值在情境切🄵時由系統快取，因此精准度較低。

可用性: macOS >= 10.12.

在 3.13 版被加入.

以下常數是唯一可以傳遞給*clock_settime()* 的參數。

time.**CLOCK_REALTIME**

即時時鐘。設定此時鐘需要適當的權限。時鐘對於所有行程都會是相同。

可用性: Unix.

在 3.3 版被加入.

16.3.3 時區常數

time.**altzone**

如果本地 DST 時區有被定義，則此值🄵本地 DST 時區相對於 UTC 以西的偏移量（以秒🄵單位）。若本地 DST 時區位於 UTC 以東（例如包括英國在🄵的西歐），則此值🄵負值。僅在 daylight 🄵非零時使用此值。詳情請參見下方🄵釋。

time.**daylight**

如果定義了 DST 時區，則🄵非零值。詳情請參見下方🄵釋。

time.**timezone**

本地（非 DST）時區相對於 UTC 以西的偏移量（以秒🄵單位），西歐大多數地區🄵負，美國🄵正，英國🄵零。詳情請參見下方🄵釋。

time.**tzname**

一個包含兩個字串的元組：第一個是本地非 DST 時區的名稱，第二個是本地 DST 時區的名稱。如果🄵有定義 DST 時區，則不應使用第二個字串。詳情請參見下方🄵釋。

備🄵

對於上述時區常數（*altzone*、*daylight*、*timezone* 和 *tzname*），其值由模組載入時或 *tzset()* 最後一次被呼叫時的時區規則🄵定，且過去的時間可能會不准確。建議使用 *localtime()* 回傳的 *tm_gmtoff* 和 *tm_zone* 來取得時區資訊。

也參考

***datetime* 模組**
更多物件導向的日期和時間介面。

***locale* 模組**
國際化服務。區域設定會影響 *strftime()* 和 *strptime()* 中許多格式指定符號 (format specifiers) 的解譯。

***calendar* 模組**
通用的日🄵相關函式。*timegm()* 是本模組中 *gmtime()* 的反函式。

🄵解

16.4 logging --- Python 的日🄵記🄵工具

原始碼： Lib/logging/__init__.py

Important

此頁面包含 API 參考訊息。有關教學流程和更進階的主題討論，請參🄵

- 基礎教學
- 進階教學
- 日🄵記🄵手🄵

這個模組定義了函式與類🄵 (class)，🄵應用程式和函式庫實作彈性的日🄵管理系統。

由標准函式庫模組提供的日🄵記🄵 API 的主要好處是，所有的 Python 模組都能參與日🄵記🄵，因此你的應用程式日🄵可以包含你自己的訊息，🄵與來自第三方模組的訊息整合在一起。

Here's a simple example of idiomatic usage:

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

The key feature of this idiomatic usage is that the majority of code is simply creating a module level logger with `getLogger(__name__)`, and using that logger to do any needed logging. This is concise, while allowing downstream code fine-grained control if needed. Logged messages to the module-level logger get forwarded to handlers of loggers in higher-level modules, all the way up to the highest-level logger known as the root logger; this approach is known as hierarchical logging.

For logging to be useful, it needs to be configured: setting the levels and destinations for each logger, potentially changing how specific modules log, often based on command-line arguments or application configuration. In most cases, like the one above, only the root logger needs to be so configured, since all the lower level loggers at module level eventually forward their messages to its handlers. *`basicConfig()`* provides a quick way to configure the root logger that handles many use cases.

這個模組提供了很多的功能性以及彈性。如果你對於 logging 不熟悉，熟悉它最好的方法就是去看教學（**請看右上方的連結**）。

The basic classes defined by the module, together with their attributes and methods, are listed in the sections below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- 格式器指定日🄵記🄵在最終輸出中的🄵局。

16.4.1 Logger 物件

Loggers have the following attributes and methods. Note that Loggers should *NEVER* be instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to *getLogger()* with the same name will always return a reference to the same Logger object.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. In addition, all loggers are descendants of the root logger. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That's because in a module, `__name__` is the module's name in the Python package namespace.

class logging.**Logger**

name

This is the logger's name, and is the value that was passed to *getLogger()* to obtain the logger.

> **備🄵**
>
> This attribute should be treated as read-only.

level

The threshold of this logger, as set by the *setLevel()* method.

> **備🄵**
>
> Do not set this attribute directly - always use *setLevel()*, which has checks for the level passed to it.

parent

The parent logger of this logger. It may change based on later instantiation of loggers which are higher up in the namespace hierarchy.

> **備🄵**
>
> This value should be treated as read-only.

propagate

如果此屬性評估🄵 true，則在此日🄵紀🄵器被記🄵的事件會被傳到更高階（上代）日🄵記🄵器的處理函式和所有附加在此日🄵記🄵器的任何處理器。訊息會直接傳到上代 loggers 的處理器 - 在問題中上代日🄵記🄵器的層級或是篩選器都不會被考慮。

If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger's level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

此建構函式將該屬性設🄵 `True`。

> **備🄵**
>
> If you attach a handler to a logger *and* one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to `True`. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

handlers

The list of handlers directly attached to this logger instance.

> **備🄵**
>
> This attribute should be treated as read-only; it is normally changed via the *addHandler()* and *removeHandler()* methods, which use locks to ensure thread-safe operation.

disabled

This attribute disables handling of any events. It is set to `False` in the initializer, and only changed by logging configuration code.

> **備🄵**
>
> This attribute should be treated as read-only.

setLevel(*level*)

Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored; logging messages which have severity *level* or higher will be emitted by whichever handler or handlers service this logger, unless a handler's level has been set to a higher severity level than *level*.

當一個日🄵記🄵器被建立時，記🄵層級會被設定成*NOTSET*（當此日🄵記🄵器是根日🄵記🄵器，或是代表父日🄵記🄵器的非根日🄵記🄵器時，會使所有訊息被處理）。請注意根日🄵記🄵器會以記🄵等級*WARNING*被建立。

The term 'delegation to the parent' means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor's level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

層級清單請見*Logging Levels*。

在 3.2 版的變更: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as *INFO*. Note, however, that levels are internally stored as integers, and methods such as e.g. *getEffectiveLevel()* and *isEnabledFor()* will return/expect to be passed integers.

isEnabledFor(*level*)

Indicates if a message of severity *level* would be processed by this logger. This method checks first the module-level level set by `logging.disable(level)` and then the logger's effective level as determined by *getEffectiveLevel()*.

getEffectiveLevel()

Indicates the effective level for this logger. If a value other than *NOTSET* has been set using *setLevel()*, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than *NOTSET* is found, and that value is returned. The value returned is an integer, typically one of *logging.DEBUG*, *logging.INFO* etc.

getChild(*suffix*)

Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string.

在 3.2 版被加入.

getChildren()

Returns a set of loggers which are immediate children of this logger. So for example `logging.getLogger().getChildren()` might return a set containing loggers named `foo` and `bar`, but a logger named `foo.bar` wouldn't be included in the set. Likewise, `logging.getLogger('foo').getChildren()` might return a set including a logger named `foo.bar`, but it wouldn't include one named `foo.bar.baz`.

在 3.12 版被加入.

debug(*msg*, **args*, ***kwargs*)

Logs a message with level *DEBUG* on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.) No % formatting operation is performed on *msg* when no *args* are supplied.

There are four keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, *stacklevel* and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) or an exception instance is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is *stacklevel*, which defaults to `1`. If greater than 1, the corresponding number of stack frames are skipped when computing the line number and function name set in the *LogRecord* created for the logging event. This can be used in logging helpers so that the function name, filename and line number recorded are not the information for the helper function/method, but rather its caller. The name of this parameter mirrors the equivalent one in the *warnings* module.

The fourth keyword argument is *extra* which can be used to pass a dictionary which is used to populate the `__dict__` of the *LogRecord* created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the section on *LogRecord* 屬性 for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the `LogRecord`. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatter`s would be used with particular `Handler`s.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant `Logger.propagate` attributes), the message will be sent to the handler set on `lastResort`.

在 3.2 版的變更: 新增 *stack_info* 參數。

在 3.5 版的變更: The *exc_info* parameter can now accept exception instances.

在 3.8 版的變更: 新增 *stacklevel* 參數。

info(*msg*, **args*, ***kwargs*)

Logs a message with level `INFO` on this logger. The arguments are interpreted as for `debug()`.

warning(*msg*, **args*, ***kwargs*)

在此記𠘨器上記𠘨一條層級𠘨`WARNING` 的訊息。這些引數被直譯的方式與`debug()` 相同。

> **備𠘨**
>
> There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

error(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`.

critical(*msg*, **args*, ***kwargs*)

Logs a message with level `CRITICAL` on this logger. The arguments are interpreted as for `debug()`.

log(*level*, *msg*, **args*, ***kwargs*)

Logs a message with integer level *level* on this logger. The other arguments are interpreted as for `debug()`.

exception(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter(*filter*)

在該 logger 🄵增加指定的 filter *filter*。

removeFilter(*filter*)

在該 logger 🄵移除指定的 filter *filter*。

filter(*record*)

Apply this logger's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)

Adds the specified handler *hdlr* to this logger.

removeHandler(*hdlr*)

Removes the specified handler *hdlr* from this logger.

findCaller(*stack_info=False*, *stacklevel=1*)

Finds the caller's source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless *stack_info* is `True`.

The *stacklevel* parameter is passed from code calling the *debug()* and other APIs. If greater than 1, the excess is used to skip stack frames before determining the values to be returned. This will generally be useful when calling logging APIs from helper/wrapper code, so that the information in the event log refers not to the helper/wrapper code, but to the code that calls it.

handle(*record*)

Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using *filter()*.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized *LogRecord* instances.

hasHandlers()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the 'propagate' attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

在 3.2 版被加入.

在 3.7 版的變更: Loggers can now be pickled and unpickled.

16.4.2 Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value	What it means / When to use it
`logging.`**`NOTSET`**	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to `NOTSET`, then all events are logged. When set on a handler, all events are handled.
`logging.`**`DEBUG`**	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
`logging.`**`INFO`**	20	Confirmation that things are working as expected.
`logging.`**`WARNING`**	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
`logging.`**`ERROR`**	40	Due to a more serious problem, the software has not been able to perform some function.
`logging.`**`CRITICAL`**	50	A serious error, indicating that the program itself may be unable to continue running.

16.4.3 Handler Objects

Handlers have the following attributes and methods. Note that *`Handler`* is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call *`Handler.__init__()`*.

class `logging.`**`Handler`**

`__init__`(*level=NOTSET*)

Initializes the *`Handler`* instance by setting its level, setting the list of filters to the empty list and creating a lock (using *`createLock()`*) for serializing access to an I/O mechanism.

`createLock`()

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

`acquire`()

Acquires the thread lock created with *`createLock()`*.

`release`()

Releases the thread lock acquired with *`acquire()`*.

`setLevel`(*level*)

Sets the threshold for this handler to *level*. Logging messages which are less severe than *level* will be ignored. When a handler is created, the level is set to *`NOTSET`* (which causes all messages to be processed).

層級清單請見*Logging Levels*。

在 3.2 版的變更: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as *`INFO`*.

`setFormatter`(*fmt*)

Sets the formatter for this handler to *fmt*. The *fmt* argument must be a *`Formatter`* instance or `None`.

addFilter(*filter*)

Adds the specified filter *filter* to this handler.

removeFilter(*filter*)

Removes the specified filter *filter* from this handler.

filter(*record*)

Apply this handler's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

close()

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal map of handlers, which is used for handler lookup by name.

Subclasses should ensure that this gets called from overridden *close()* methods.

handle(*record*)

Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

handleError(*record*)

This method should be called from handlers when an exception is encountered during an *emit()* call. If the module-level attribute *raiseExceptions* is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of *raiseExceptions* is `True`, as that is more useful during development).

format(*record*)

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

emit(*record*)

Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a *NotImplementedError*.

> **警告**
>
> This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically:
>
> - Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.
> - Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the module-level lock *after* the handler-level lock (because in this method, the handler-level lock has already been acquired).

For a list of handlers included as standard, see *logging.handlers*.

16.4.4 Formatter Objects

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, *, *defaults=None*)

Responsible for converting a *LogRecord* to an output string to be interpreted by a human or external system.

參數

- **fmt** (str) -- A format string in the given *style* for the logged output as a whole. The possible mapping keys are drawn from the *LogRecord* object's *LogRecord 屬性*. If not specified, '%(message)s' is used, which is just the logged message.
- **datefmt** (str) -- A format string for the date/time portion of the logged output. If not specified, the default described in *formatTime()* is used.
- **style** (str) -- Can be one of '%', '{' or '$' and determines how the format string will be merged with its data: using one of *printf-style String Formatting* (%), *str.format()* ({) or *string.Template* ($). This only applies to *fmt* (e.g. '%(message)s' versus '{message}'), not to the actual log messages passed to the logging methods. However, there are other ways to use {- and $-formatting for log messages.
- **validate** (bool) -- If True (the default), incorrect or mismatched *fmt* and *style* will raise a *ValueError*; for example, logging.Formatter('%(asctime)s - %(message)s', style='{').
- **defaults** (dict[str, *Any*]) -- A dictionary with default values to use in custom fields. For example, logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})

在 3.2 版的變更: 新增 *style* 參數。

在 3.8 版的變更: 新增 *validate* 參數。

在 3.10 版的變更: 新增 *defaults* 參數。

format(*record*)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The *message* attribute of the record is computed using *msg % args*. If the formatting string contains '(asctime)', *formatTime()* is called to format the event time. If there is exception information, it is formatted using *formatException()* and appended to the message. Note that the formatted exception information is cached in attribute *exc_text*. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one *Formatter* subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value (by setting the *exc_text* attribute to None) after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value, but recalculates it afresh.

If stack information is available, it's appended after the exception information, using *formatStack()* to transform it if necessary.

formatTime(*record*, *datefmt=None*)

This method should be called from *format()* by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if *datefmt* (a string) is specified, it is used with *time.strftime()* to format the creation time of the record. Otherwise, the format '%Y-%m-%d %H:%M:%S,uuu' is used, where the uuu part is a millisecond value and the other letters are as per the *time.strftime()* documentation. An example time in this format is 2003-01-23 00:29:50,411. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, *time.localtime()* is used; to change this for a particular formatter instance, set the converter attribute to a function with the same signature as *time.localtime()* or *time.gmtime()*. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the converter attribute in the Formatter class.

在 3.3 版的變更: Previously, the default format was hard-coded as in this example: 2010-09-06 22:38:15,292 where the part before the comma is handled by a strptime format string ('%Y-%m-%d

`%H:%M:%S`'), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s, %03d'` --- and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and `default_msec_format` (for appending the millisecond value).

在 3.9 版的變更: The `default_msec_format` can be `None`.

formatException(*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by *sys.exc_info()*) as a string. This default implementation just uses *traceback.print_exception()*. The resulting string is returned.

formatStack(*stack_info*)

Formats the specified stack information (a string as returned by *traceback.print_stack()*, but with the last newline removed) as a string. This default implementation just returns the input value.

class logging.**BufferingFormatter**(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a *Formatter* instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

16.4.5 Filter Objects

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.**Filter**(*name=''*)

Returns an instance of the *Filter* class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)

Is the specified record to be logged? Returns false for no, true for yes. Filters can either modify log records in-place or return a completely different record instance which will replace the original log record in any future processing of the event.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using *debug()*, *info()*, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter`: you can pass any instance which has a `filter` method with the same semantics.

在 3.2 版的變更: You don't need to create specialized `Filter` classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its *`filter()`* method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by *`filter()`*.

在 3.12 版的變更: You can now return a *`LogRecord`* instance from filters to replace the log record rather than modifying it in place. This allows filters attached to a *`Handler`* to modify the log record before it is emitted, without having side effects on other handlers.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the *`LogRecord`* being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

16.4.6 LogRecord 物件

`LogRecord` instances are created automatically by the *`Logger`* every time something is logged, and can be created manually via *`makeLogRecord()`* (for example, from a pickled event received over the wire).

class `logging.`**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in *msg* and *args*, which are combined using `msg % args` to create the `message` attribute of the record.

參數

- **name** (`str`) -- The name of the logger used to log the event represented by this `LogRecord`. Note that the logger name in the `LogRecord` will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** (`int`) -- The *numeric level* of the logging event (such as `10` for `DEBUG`, `20` for `INFO`, etc). Note that this is converted to *two* attributes of the LogRecord: `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** (`str`) -- The full string path of the source file where the logging call was made.
- **lineno** (`int`) -- The line number in the source file where the logging call was made.
- **msg** (`Any`) -- The event description message, which can be a %-format string with placeholders for variable data, or an arbitrary object (see arbitrary-object-messages).
- **args** (`tuple | dict[str, Any]`) -- Variable data to merge into the *msg* argument to obtain the event description.
- **exc_info** (`tuple[type[BaseException], BaseException, types.TracebackType] | None`) -- An exception tuple with the current exception information, as returned by *`sys.exc_info()`*, or `None` if no exception information is available.
- **func** (`str | None`) -- The name of the function or method from which the logging call was invoked.
- **sinfo** (`str | None`) -- A text string representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage()

Returns the message for this *`LogRecord`* instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, *`str()`* is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

在 3.2 版的變更: The creation of a *LogRecord* has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a *LogRecord* at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.4.7 LogRecord 屬性

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (*str.format()*), you can use `{attrname}` as the placeholder in the format string. If you are using $-formatting (*string.Template*), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of `{msecs:03.0f}` would format a millisecond value of `4` as `004`. Refer to the *str.format()* documentation for full details on the options available to you.

屬性名徵	格式	描述
args	你不應該需要自己格式化它。	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the *LogRecord* was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the *LogRecord* was created (as returned by *time.time_ns()* / 1e9).
exc_info	你不應該需要自己格式化它。	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)`	`pathname` 的檔案名稱部分。
funcName	`%(funcName)`	Name of function containing the logging call.
levelname	`%(levelname`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when *Formatter.format()* is invoked.
模組	`%(module)s`	模組（`filename` 的名稱部分）。
msecs	`%(msecs)d`	Millisecond portion of the time when the *LogRecord* was created.
msg	你不應該需要自己格式化它。	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see arbitrary-object-messages).
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	行程 ID（如果可用）。
processName	`%(processNa`	行程名稱（如果可用）。
relativeCreated	`%(relativeC`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	你不應該需要自己格式化它。	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	執行緒 ID（如果可用）。
threadName	`%(threadNam`	Thread name (if available).
taskName	`%(taskName)`	*asyncio.Task* name (if available).

在 3.1 版的變更: 新增 *processName*。

在 3.12 版的變更: 新增 *taskName*。

16.4.8 LoggerAdapter 物件

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class logging.**LoggerAdapter**(*logger*, *extra*, *merge_extra=False*)

Returns an instance of *LoggerAdapter* initialized with an underlying *Logger* instance, a dict-like object (*extra*), and a boolean (*merge_extra*) indicating whether or not the *extra* argument of individual log calls should be merged with the *LoggerAdapter* extra. The default behavior is to ignore the *extra* argument of individual log calls and only use the one of the *LoggerAdapter* instance

process(*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

manager

Delegates to the underlying `manager` on *logger*.

_log

Delegates to the underlying `_log()` method on *logger*.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger*: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably.

在 3.2 版的變更: The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

在 3.6 版的變更: Attribute `manager` and method `_log()` were added, which delegate to the underlying logger and allow adapters to be nested.

在 3.13 版的變更: 新增 *merge_extra* 引數。

16.4.9 執行緒安全

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this through using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

16.4.10 模組層級函式

In addition to the classes described above, there are a number of module-level functions.

logging.**getLogger**(*name=None*)

Return a logger with the specified name or, if name is `None`, return the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like *'a'*, *'a.b'* or *'a.b.c.d'*. Choice of these names is entirely up to the developer who is using logging, though it is recommended that `__name__` be used unless you have a specific reason for not doing that, as mentioned in *Logger 物件*.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.**getLoggerClass**()

Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customized *Logger* class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... 在這F覆蓋其行F
```

logging.**getLogRecordFactory**()

Return a callable which is used to create a *LogRecord*.

在 3.2 版被加入: This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

See *setLogRecordFactory()* for more information about the how the factory is called.

logging.**debug**(*msg*, **args*, ***kwargs*)

This is a convenience function that calls *Logger.debug()*, on the root logger. The handling of the arguments is in every way identical to what is described in that method.

The only difference is that if the root logger has no handlers, then *basicConfig()* is called, prior to calling `debug` on the root logger.

For very short scripts or quick demonstrations of `logging` facilities, `debug` and the other module-level functions may be convenient. However, most programs will want to carefully and explicitly control the logging configuration, and should therefore prefer creating a module-level logger and calling *Logger.debug()* (or other level-specific methods) on it, as described at the beginning of this documentation.

logging.**info**(*msg*, **args*, ***kwargs*)

Logs a message with level *INFO* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**warning**(*msg*, **args*, ***kwargs*)

Logs a message with level *WARNING* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

> **備F**
>
> There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

logging.**error**(*msg*, **args*, ***kwargs*)

Logs a message with level *ERROR* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**critical**(*msg*, **args*, ***kwargs*)

Logs a message with level *CRITICAL* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**exception**(*msg*, **args*, ***kwargs*)

Logs a message with level *ERROR* on the root logger. The arguments and behavior are otherwise the same as for *debug()*. Exception info is added to the logging message. This function should only be called from an exception handler.

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

Logs a message with level *level* on the root logger. The arguments and behavior are otherwise the same as for *debug()*.

logging.**disable**(*level=CRITICAL*)

Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is

called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

在 3.7 版的變更: The *level* parameter was defaulted to level `CRITICAL`. See bpo-28524 for more information about this change.

`logging.`**`addLevelName`**`(`*level*, *levelName*`)`

Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *`Formatter`* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

> **備🄵**
>
> If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.`**`getLevelNamesMapping`**`()`

Returns a mapping from level names to their corresponding logging levels. For example, the string "CRITICAL" maps to *`CRITICAL`*. The returned mapping is copied from an internal mapping on each call to this function.

在 3.11 版被加入.

`logging.`**`getLevelName`**`(`*level*`)`

Returns the textual or numeric representation of logging level *level*.

If *level* is one of the predefined levels *`CRITICAL`*, *`ERROR`*, *`WARNING`*, *`INFO`* or *`DEBUG`* then you get the corresponding string. If you have associated levels with names using *`addLevelName()`* then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

The *level* parameter also accepts a string representation of the level such as 'INFO'. In such cases, this functions returns the corresponding numeric value of the level.

If no matching numeric or string value is passed in, the string 'Level %s' % level is returned.

> **備🄵**
>
> Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *LogRecord 屬性*), and vice versa.

在 3.4 版的變更: In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`logging.`**`getHandlerByName`**`(`*name*`)`

Returns a handler with the specified *name*, or `None` if there is no handler with that name.

在 3.12 版被加入.

`logging.`**`getHandlerNames`**`()`

Returns an immutable set of all known handler names.

在 3.12 版被加入.

logging.**makeLogRecord**(*attrdict*)

Creates and returns a new *LogRecord* instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled *LogRecord* attribute dictionary, sent over a socket, and reconstituting it as a *LogRecord* instance at the receiving end.

logging.**basicConfig**(***kwargs*)

Does basic configuration for the logging system by creating a *StreamHandler* with a default *Formatter* and adding it to the root logger. The functions *debug()*, *info()*, *warning()*, *error()* and *critical()* will call *basicConfig()* automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured, unless the keyword argument *force* is set to `True`.

> **備🄵**
>
> This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

支援以下的關鍵字引數。

格式	描述
filename	Specifies that a *FileHandler* be created, using the specified filename, rather than a *StreamHandler*.
filemode	If *filename* is specified, open the file in this *mode*. Defaults to `'a'`.
format	Use the specified format string for the handler. Defaults to attributes `levelname`, `name` and `message` separated by colons.
datefmt	Use the specified date/time format, as accepted by *time.strftime()*.
style	If *format* is specified, use this style for the format string. One of `'%'`, `'{'` or `'$'` for *printf-style*, *str.format()* or *string.Template* respectively. Defaults to `'%'`.
level	Set the root logger level to the specified *level*.
stream	Use the specified stream to initialize the *StreamHandler*. Note that this argument is incompatible with *filename* - if both are present, a `ValueError` is raised.
handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* - if both are present, a `ValueError` is raised.
force	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.
encoding	If this keyword argument is specified along with *filename*, its value is used when the *FileHandler* is created, and thus used when opening the output file.
errors	If this keyword argument is specified along with *filename*, its value is used when the *FileHandler* is created, and thus used when opening the output file. If not specified, the value 'backslashreplace' is used. Note that if `None` is specified, it will be passed as such to *open()*, which means that it will be treated the same as passing 'errors'.

在 3.2 版的變更: 新增 *style* 引數。

在 3.3 版的變更: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

在 3.8 版的變更: 新增 *force* 引數。

在 3.9 版的變更: 新增 *encoding* 與 *errors* 引數。

logging.**shutdown**()

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

When the logging module is imported, it registers this function as an exit handler (see *atexit*), so normally there's no need to do that manually.

logging.**setLoggerClass**(*klass*)

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the *logging.getLogger()* API to get your loggers.

logging.**setLogRecordFactory**(*factory*)

Set a callable which is used to create a *LogRecord*.

參數

factory -- The factory callable to be used to instantiate a log record.

在 3.2 版被加入: This function has been provided, along with *getLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

The factory has the following signature:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

name
The logger name.

level
The logging level (numeric).

fn
The full pathname of the file where the logging call was made.

lno
The line number in the file where the logging call was made.

msg
The logging message.

args
The arguments for the logging message.

exc_info
An exception tuple, or `None`.

func
The name of the function or method which invoked the logging call.

sinfo
A stack traceback such as is provided by *traceback.print_stack()*, showing the call hierarchy.

kwargs
額外的關鍵字引數。

16.4.11 模組層級屬性

logging.**lastResort**

A "handler of last resort" is available through this attribute. This is a *StreamHandler* writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that "no

handlers could be found for logger XYZ". If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

在 3.2 版被加入.

`logging.`**`raiseExceptions`**

Used to see if exceptions during handling should be propagated.

Default: `True`.

If *`raiseExceptions`* is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors.

16.4.12 Integration with the warnings module

The *`captureWarnings()`* function can be used to integrate *`logging`* with the *`warnings`* module.

`logging.`**`captureWarnings`**(*capture*)

This function is used to turn the capture of warnings by logging on and off.

If *capture* is `True`, warnings issued by the *`warnings`* module will be redirected to the logging system. Specifically, a warning will be formatted using *`warnings.formatwarning()`* and the resulting string logged to a logger named `'py.warnings'` with a severity of *`WARNING`*.

If *capture* is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

也參考

***`logging.config`* 模組**
Configuration API for the logging module.

***`logging.handlers`* 模組**
Useful handlers included with the logging module.

PEP 282 - A Logging System
The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package
This is the original source for the *`logging`* package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the *`logging`* package in the standard library.

16.5 `logging.config` --- 日🄵記🄵配置

原始碼： Lib/logging/config.py

Important

This page contains only reference information. For tutorials, please see

- 基礎教學
- 進階教學
- Logging Cookbook

This section describes the API for configuring the logging module.

16.5.1 Configuration functions

The following functions configure the logging module. They are located in the *logging.config* module. Their use is optional --- you can configure the logging module using these functions or by making calls to the main API (defined in *logging* itself) and defining handlers which are declared either in *logging* or *logging.handlers*.

logging.config.**dictConfig**(*config*)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in *Configuration dictionary schema* below.

If an error is encountered during configuration, this function will raise a *ValueError*, *TypeError*, *AttributeError* or *ImportError* with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The *logging.config* module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

dictConfig() calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then *dictConfig()* could be called exactly as in the default, uncustomized state.

在 3.2 版被加入.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*, *encoding=None*)

Reads the logging configuration from a *configparser*-format file. The format of the file should be as described in *Configuration file format*. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

It will raise *FileNotFoundError* if the file doesn't exist and *RuntimeError* if the file is invalid or empty.

參數

- **fname** -- A filename, or a file-like object, or an instance derived from *RawConfigParser*. If a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a *ConfigParser* is instantiated, and the configuration read by it from the object passed in `fname`. If that has a *readline()* method, it is assumed to be a file-like object and read using *read_file()*; otherwise, it is assumed to be a filename and passed to *read()*.
- **defaults** -- Defaults to be passed to the `ConfigParser` can be specified in this argument.
- **disable_existing_loggers** -- If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in

a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.

- **encoding** -- The encoding used to open file when *fname* is filename.

在 3.4 版的變更: An instance of a subclass of *RawConfigParser* is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

在 3.10 版的變更: Added the *encoding* parameter.

在 3.12 版的變更: An exception will be thrown if the provided file doesn't exist or is invalid or empty.

`logging.config.`**`listen`**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by *dictConfig()* or *fileConfig()*. Returns a *Thread* instance on which you can call *start()* to start the server, and which you can *join()* when appropriate. To stop the server, call *stopListening()*.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

備🄵

Because portions of the configuration are passed through *eval()*, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls *listen()*. Specifically, if the process calling *listen()* runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's *listen()* socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the `verify` argument to *listen()* to prevent unrecognised configurations from being applied.

在 3.4 版的變更: 新增 `verify` 引數。

備🄵

If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use *dictConfig()* for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

`logging.config.`**`stopListening`**()

Stops the listening server which was created with a call to *listen()*. This is typically called before calling `join()` on the return value from *listen()*.

16.5.2 Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in *User-defined objects*. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.5.3 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to `dictConfig()` must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `Formatter` instance.

  The configuring dict is searched for the following optional keys which correspond to the arguments passed to create a `Formatter` object:

  - `format`
  - `datefmt`
  - `style`
  - `validate` (since version >=3.8)
  - `defaults` (since version >=3.12)

  An optional `class` key indicates the name of the formatter's class (as a dotted module and class name). The instantiation arguments are as for `Formatter`, thus this key is most useful for instantiating a customised subclass of `Formatter`. For example, the alternative class might present exception tracebacks in an expanded or condensed format. If your formatter requires different or extra configuration keys, you should use *User-defined objects*.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.

- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.

- `filters` (optional). A list of ids of the filters for this handler.

  在 3.11 版的變更: `filters` can take filter instances in addition to ids.

All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

the handler with id `console` is instantiated as a *`logging.StreamHandler`*, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a *`logging.handlers.RotatingFileHandler`* with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.

    在 3.11 版的變更: `filters` can take filter instances in addition to ids.

  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing *`fileConfig()`* API.

  If the specified value is `True`, the configuration is processed as described in the section on *Incremental Configuration*.

- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in *`fileConfig()`*. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is

problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard *`logging.Formatter`* instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a *`logging.Formatter`* initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard *`logging.Formatter`* instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

⚠ 警告

The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

在 3.11 版的變更: The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

> **警告**
>
> The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured *after* any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the *`logging`* module. For example, consider *`logging.handlers.MemoryHandler`*, which takes a `target` argument which is another

handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team@domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. Please note that the characters `[` and `]` are not allowed in the keys. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin *`__import__()`* function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with *`staticmethod()`*. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with *`staticmethod()`* if you're setting the import callable on a configurator *instance*.

Configuring QueueHandler and QueueListener

If you want to configure a *QueueHandler*, noting that this is normally used in conjunction with a *QueueListener*, you can configure both together. After the configuration, the QueueListener instance will be available as the *listener* attribute of the created handler, and that in turn will be available to you using *getHandlerByName()* and passing the name you have used for the QueueHandler in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The queue and listener keys are optional.

If the queue key is present, the corresponding value can be one of the following:

- An object implementing the *Queue.put_nowait* and *Queue.get* public API. For instance, this may be an actual instance of *queue.Queue* or a subclass thereof, or a proxy obtained by *multiprocessing.managers.SyncManager.Queue()*.

  This is of course only possible if you are constructing or modifying the configuration dictionary in code.

- A string that resolves to a callable which, when called with no arguments, returns the queue instance to use. That callable could be a *queue.Queue* subclass or a function which returns a suitable queue instance, such as my.module.queue_factory().

- A dict with a '()' key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a *queue.Queue* instance.

If the queue key is absent, a standard unbounded *queue.Queue* instance is created and used.

If the listener key is present, the corresponding value can be one of the following:

- A subclass of *logging.handlers.QueueListener*. This is of course only possible if you are constructing or modifying the configuration dictionary in code.

- A string which resolves to a class which is a subclass of QueueListener, such as 'my.package.CustomListener'.

- A dict with a '()' key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a callable with the same signature as the QueueListener initializer.

If the listener key is absent, *logging.handlers.QueueListener* is used.

The values under the handlers key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as *QueueHandler* and *QueueListener*.

在 3.12 版被加入.

16.5.4 Configuration file format

The configuration file format understood by *fileConfig()* is based on *configparser* functionality. The file must contain sections called [loggers], [handlers] and [formatters] which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named log01 in the [loggers] section, the relevant configuration details are held in a section [logger_log01]. Similarly, a handler called hand01 in the [handlers] section will have its configuration held in a section called [handler_hand01], while a formatter called form01 in the [formatters]

section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

> **備𝔽**
>
> The *`fileConfig()`* API is older than the *`dictConfig()`* API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure *`Filter`* objects, which provide for filtering of messages beyond simple integer levels, using *`fileConfig()`*. If you need to have instances of *`Filter`* in your logging configuration, you will need to use *`dictConfig()`*. Note that future enhancements to configuration functionality will be added to *`dictConfig()`*, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are *evaluated* in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by *`eval()`* in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when *evaluated* in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when *evaluated* in the context of the `logging` package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema *formatters section*.

The `defaults` entry, when *evaluated* in the context of the `logging` package's namespace, is a dictionary of default values for custom formatting fields. If not provided, it defaults to `None`.

備🄵

Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen()` documentation for more information.

也參考

***logging* 模組**
API reference for the logging module.

***logging.handlers* 模組**
Useful handlers included with the logging module.

16.6 `logging.handlers` --- 日🄵紀🄵處理器

原始碼： Lib/logging/handlers.py

Important

This page contains only reference information. For tutorials, please see

- 基礎教學
- 進階教學
- Logging Cookbook

The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.

16.6.1 StreamHandler

The `StreamHandler` class, located in the core `logging` package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).

class `logging.`**`StreamHandler`**(*stream=None*)

Returns a new instance of the `StreamHandler` class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream followed by *terminator*. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush()

Flushes the stream by calling its *flush()* method. Note that the `close()` method is inherited from *Handler* and so does no output, so an explicit *flush()* call may be needed at times.

setStream(*stream*)

Sets the instance's stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

參數

stream -- The stream that the handler should use.

回傳

the old stream, if the stream was changed, or `None` if it wasn't.

在 3.7 版被加入.

terminator

String used as the terminator when writing a formatted record to a stream. Default value is `'\n'`.

If you don't want a newline termination, you can set the handler instance's `terminator` attribute to the empty string.

In earlier versions, the terminator was hardcoded as `'\n'`.

在 3.2 版被加入.

16.6.2 FileHandler

The *FileHandler* class, located in the core *logging* package, sends logging output to a disk file. It inherits the output functionality from *StreamHandler*.

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is specified, it's used to determine how encoding errors are handled.

在 3.6 版的變更: As well as string values, *Path* objects are also accepted for the *filename* argument.

在 3.9 版的變更: 新增 *errors* 參數。

close()

Closes the file.

emit(*record*)

Outputs the record to the file.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w', the record will not be emitted (see bpo-42378).

16.6.3 NullHandler

在 3.1 版被加入.

The *NullHandler* class, located in the core *logging* package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

class logging.**NullHandler**

Returns a new instance of the *NullHandler* class.

emit(*record*)

This method does nothing.

handle(*record*)

This method does nothing.

createLock()

This method returns `None` for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use *NullHandler*.

16.6.4 WatchedFileHandler

The *WatchedFileHandler* class, located in the *logging.handlers* module, is a `FileHandler` which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; *stat()* always returns zero for this value.

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *WatchedFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

在 3.6 版的變更: As well as string values, *Path* objects are also accepted for the *filename* argument.

在 3.9 版的變更: 新增 *errors* 參數。

reopenIfNeeded()

Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

在 3.6 版被加入.

emit(*record*)

Outputs the record to the file, but first calls *reopenIfNeeded()* to reopen the file if it has changed.

16.6.5 BaseRotatingHandler

The *BaseRotatingHandler* class, located in the *logging.handlers* module, is the base class for the rotating file handlers, *RotatingFileHandler* and *TimedRotatingFileHandler*. You should not need to instantiate this class, but it has attributes and methods you may need to override.

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)

The parameters are as for `FileHandler`. The attributes are:

namer

If this attribute is set to a callable, the *rotation_filename()* method delegates to this callable. The parameters passed to the callable are those passed to *rotation_filename()*.

> **備🄴**
>
> The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.
>
> It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, *RotatingFileHandler* expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and *TimedRotatingFileHandler* deletes old log files (based on the `backupCount` parameter passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of *TimedRotatingFileHandler* which overrides the *getFilesToDelete()* method to fit in with the custom naming scheme.)

在 3.3 版被加入.

rotator

If this attribute is set to a callable, the *rotate()* method delegates to this callable. The parameters passed to the callable are those passed to *rotate()*.

在 3.3 版被加入.

rotation_filename(*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the 'namer' attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is `None`), the name is returned unchanged.

參數

default_name -- The default name for the log file.

在 3.3 版被加入.

rotate(*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the 'rotator' attribute of the handler, if it's callable, passing the source and dest arguments to it. If the attribute isn't callable (the default is `None`), the source is simply renamed to the destination.

參數

- **source** -- The source filename. This is normally the base filename, e.g. 'test.log'.
- **dest** -- The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

在 3.3 版被加入.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of *RotatingFileHandler* and *TimedRotatingFileHandler*. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an `emit()` call, i.e. via the `handleError()` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see cookbook-rotator-namer.

16.6.6 RotatingFileHandler

The *RotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the *RotatingFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

在 3.6 版的變更: As well as string values, *Path* objects are also accepted for the *filename* argument.

在 3.9 版的變更: 新增 *errors* 參數。

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described previously.

shouldRollover(*record*)

See if the supplied record would cause the file to exceed the configured size limit.

16.6.7 TimedRotatingFileHandler

The *TimedRotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

Returns a new instance of the *TimedRotatingFileHandler* class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval	If/how *atTime* is used
`'S'`	Seconds	Ignored
`'M'`	Minutes	Ignored
`'H'`	Hours	Ignored
`'D'`	Days	Ignored
`'W0'-'W6'`	Weekday (0=Monday)	Used to compute initial rollover time
`'midnight'`	Roll over at midnight, if *atTime* not specified, else at time *atTime*	Used to compute initial rollover time

When using weekday-based rotation, specify 'W0' for Monday, 'W1' for Tuesday, and so on up to 'W6' for Sunday. In this case, the value passed for *interval* isn't used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to `emit()`.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen "at midnight" or "on a particular weekday". Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

If *errors* is specified, it's used to determine how encoding errors are handled.

> **備🄵**
>
> Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of "every minute" is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

在 3.4 版的變更: 新增 *atTime* 參數。

在 3.6 版的變更: As well as string values, `Path` objects are also accepted for the *filename* argument.

在 3.9 版的變更: 新增 *errors* 參數。

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described above.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These

shouldRollover(*record*)

See if enough time has passed for a rollover to occur and if it has, compute the next rollover time.

16.6.8 SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.**SocketHandler**(*host*, *port*)

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

在 3.4 版的變更: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()

Closes the socket.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

handleError()

Handles an error which has occurred during *emit()*. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (*socket.SOCK_STREAM*).

makePickle(*record*)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)

Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for *makePickle()*.

This function allows for partial sends, which can happen when the network is busy.

createSocket()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.6.9 DatagramHandler

The *DatagramHandler* class, located in the *logging.handlers* module, inherits from *SocketHandler* to support sending logging messages over UDP sockets.

class logging.handlers.**DatagramHandler**(*host*, *port*)

Returns a new instance of the *DatagramHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

> **備🄵**
>
> As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

在 3.4 版的變更: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

makeSocket()

The factory method of *SocketHandler* is here overridden to create a UDP socket (*socket.SOCK_DGRAM*).

send(*s*)

Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for *SocketHandler.makePickle()*.

16.6.10 SysLogHandler

The *SysLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*, *timeout=None*)

Returns a new instance of the *SysLogHandler* class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to *socket.SOCK_DGRAM* and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of *socket.SOCK_STREAM*. If *timeout* is specified, it sets a timeout (in seconds) for the socket operations. This can help prevent the program from hanging indefinitely if the syslog server is unreachable. By default, *timeout* is `None`, meaning no timeout is applied.

Note that if your server is not listening on UDP port 514, *SysLogHandler* may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

> **備🄵**
>
> On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect *SysLogHandler* to work on this system.
>
> See gh-91070 for more information.

在 3.2 版的變更: 新增 *socktype*。

在 3.14 版的變更: 新增 *timeout*。

close()

Closes the socket to the remote host.

createSocket()

Tries to create a socket and, if it's not a datagram socket, connect it to the other end. This method is called during handler initialization, but it's not regarded as an error if the other end isn't listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

在 3.11 版被加入.

emit(*record*)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

在 3.2.1 版的變更: (See: bpo-12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification (**RFC 5424**). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

在 3.3 版的變更: (See: bpo-12419.) In earlier versions, there was no facility for an "ident" or "tag" prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(*facility*, *priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in *`SysLogHandler`* and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` 或 `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` 或 `panic`	LOG_EMERG
`err` 或 `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` 或 `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to 'warning'.

16.6.11 NTEventLogHandler

The *`NTEventLogHandler`* class, located in the *`logging.handlers`* module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the *`NTEventLogHandler`* class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

16.6.12 SMTPHandler

The *SMTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to an email address via SMTP.

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the *SMTPHandler* class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *smtplib.SMTP.starttls()* method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

在 3.3 版的變更: 新增 *timeout* 參數。

emit(*record*)

Formats the record and sends it to the specified addressees.

getSubject(*record*)

If you want to specify a subject line which is record-dependent, override this method.

16.6.13 MemoryHandler

The *MemoryHandler* class, located in the *logging.handlers* module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general *BufferingHandler*, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

class logging.handlers.**BufferingHandler**(*capacity*)

Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)

Append the record to the buffer. If *shouldFlush()* returns true, call *flush()* to process the buffer.

flush()

For a *BufferingHandler* instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(*record*)

Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

Returns a new instance of the *MemoryHandler* class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using *setTarget()* before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

在 3.6 版的變更: 新增 *flushOnClose* 參數。

close()

Calls *flush()*, sets the target to `None` and clears the buffer.

flush()

For a *MemoryHandler* instance, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when buffered records are sent to the target. Override if you want different behavior.

setTarget(*target*)

Sets the target handler for this handler.

shouldFlush(*record*)

Checks for buffer full or a record at the *flushLevel* or higher.

16.6.14 HTTPHandler

The *HTTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to a web server, using either `GET` or `POST` semantics.

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the *HTTPHandler* class. The *host* can be of the form `host:port`, should you need to use a specific port number. If no *method* is specified, `GET` is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a *ssl.SSLContext* instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP 'Authorization' header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

在 3.5 版的變更: 新增 *context* 參數。

mapLogRecord(*record*)

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of *LogRecord* is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(*record*)

Sends the record to the web server as a URL-encoded dictionary. The *mapLogRecord()* method is used to convert the record to the dictionary to be sent.

> **備🄵**
>
> Since preparing a record for sending it to a web server is not the same as a generic formatting operation, using *setFormatter()* to specify a *Formatter* for a *HTTPHandler* has no effect. Instead of calling *format()*, this handler calls *mapLogRecord()* and then *urllib.parse.urlencode()* to encode the dictionary in a form suitable for sending to a web server.

16.6.15 QueueHandler

在 3.2 版被加入.

The *QueueHandler* class, located in the *logging.handlers* module, supports sending logging messages to a queue, such as those implemented in the *queue* or *multiprocessing* modules.

Along with the *QueueListener* class, *QueueHandler* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueHandler**(*queue*)

Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use *SimpleQueue* instances for *queue*.

> **備🄵**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

> **警告**
>
> The *multiprocessing* module uses an internal logger created and accessed via *get_logger()*. *multiprocessing.Queue* will log DEBUG level messages upon items being queued. If those log messages are processed by a *QueueHandler* using the same *multiprocessing.Queue* instance, it will cause a deadlock or infinite recursion.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the *handleError()* method is called to handle the error. This can result in the record silently being dropped (if *logging.raiseExceptions* is False) or a message printed to sys.stderr (if *logging.raiseExceptions* is True).

prepare(*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's msg and message attributes with the merged message (obtained by calling the handler's *format()* method), and sets the args, exc_info and exc_text attributes to None.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

> **備🄵**
>
> The base implementation formats the message with arguments, sets the message and msg attributes to the formatted message and sets the args and exc_text attributes to None to allow pickling and to prevent further attempts at formatting. This means that a handler on the *QueueListener* side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass QueueHandler and override this method to e.g. avoid setting exc_text to None. Note that the message / msg / args changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your args are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(*record*)

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

listener

When created via configuration using *dictConfig()*, this attribute will contain a *QueueListener* instance for use with this handler. Otherwise, it will be `None`.

在 3.12 版被加入.

16.6.16 QueueListener

在 3.2 版被加入.

The *QueueListener* class, located in the *logging.handlers* module, supports receiving logging messages from a queue, such as those implemented in the *queue* or *multiprocessing* modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While *QueueListener* is not itself a handler, it is documented here because it works hand-in-hand with *QueueHandler*.

Along with the *QueueHandler* class, *QueueListener* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the *QueueListener* class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the *dequeue()* method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use *SimpleQueue* instances for *queue*.

> **備🄵**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

在 3.5 版的變更: 新增 `respect_handler_level` 引數。

在 3.14 版的變更: *QueueListener* can now be used as a context manager via `with`. When entering the context, the listener is started. When exiting the context, the listener is stopped. *__enter__()* returns the *QueueListener* object.

dequeue(*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from *prepare()*.

start()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

在 3.14 版的變更: Raises *RuntimeError* if called and the listener is already running.

stop()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

在 3.3 版被加入.

也參考

***logging* 模組**
API reference for the logging module.

***logging.config* 模組**
Configuration API for the logging module.

16.7 `platform` --- 對底層平台識🄵資料的存取

原始碼： Lib/platform.py

備🄵

特定平臺清單🄵按字母順序排列，Linux 包括在 Unix 小節之中。

16.7.1 跨平台

`platform.`**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)

查詢給定的可執行檔案（預設🄵 Python 直譯器二進位制檔案）來取得各種架構資訊。

回傳一個 tuple（元組）`(bits, linkage)`，其中包含可執行檔案所使用的位元架構和連結格式資訊。這兩個值均以字串形式回傳。

無法確定的值將回傳參數所給定之預先設置值。如果給定的位元🄵 `''`，則會使用 `sizeof(pointer)`（或者當 Python 版本 < 1.5.2 時🄵 `sizeof(long)`）作🄵所支援指標大小的指示器 (indicator)。

此函式依賴於系統的 `file` 命令來執行實際的操作。這在幾乎所有 Unix 平臺和某些非 Unix 平臺上，只有當可執行檔案指向 Python 直譯器時才可使用。當以上要求不滿足時將會使用合理的預設值。

備🄵

在 macOS（也許還有其他平臺）上，可執行檔案可能是包含多種架構的通用檔案。

要取得目前直譯器的"64 位元性 (64-bitness)"，更可靠的做法是查詢*sys.maxsize* 屬性：

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**()

回傳機器種類，例如 'AMD64'。如果該值無法確定則會回傳一個空字串。

platform.**node**()

回傳電腦的網路名稱（可能不是完整名稱!）。如果該值無法確定則會回傳一個空字串。

platform.**platform**(*aliased=False*, *terse=False*)

會🄵可能附帶有用資訊地回傳一個標識底層平臺的字串。

輸出應🄵人類易讀的 *(human readable)*，而非機器易剖析的 (machine parseable)。它在不同平臺上看起來可能不一致，這是有意🄵之的。

如果 *aliased* 🄵真值，此函式將使用各種不同於平臺通用名稱的🄵名來回報系統名稱，例如 SunOS 將被回報🄵 Solaris。*system_alias()* 函式被用於實作此功能。

將 *terse* 設🄵真值將導致此函式只回傳標識平臺所需的最小量資訊。

在 3.8 版的變更: 在 macOS 上，如果*mac_ver()* 回傳的釋出版字串非空字串，此函式現在會使用它以取得 macOS 版本而非 darwin 版本。

platform.**processor**()

回傳（真實的）處理器名稱，例如 'amdk6'。

如果該值無法確定則將回傳空字串。請注意，許多平臺都不提供此資訊或是簡單地回傳與*machine()* 相同的值。NetBSD 則會提供此資訊。

platform.**python_build**()

回傳一個 tuple (buildno, builddate)，表示字串形式的 Python 建置編號和日期。

platform.**python_compiler**()

回傳一個標識用於編譯 Python 的編譯器的字串。

platform.**python_branch**()

回傳一個標識 Python 實作 SCM 分支的字串。

platform.**python_implementation**()

回傳一個標識 Python 實作的字串。可能的回傳值有：'CPython'、'IronPython'、'Jython'、'PyPy'。

platform.**python_revision**()

回傳一個標識 Python 實作 SCM 修訂版的字串。

platform.**python_version**()

將 Python 版本以字串 'major.minor.patchlevel' 形式回傳。

請注意此回傳值不同於 Python sys.version，它總是會包括 patchlevel（預設🄵 '0'）。

platform.**python_version_tuple**()

將 Python 版本以字串 tuple (major, minor, patchlevel) 形式回傳。

請注意此回傳值不同於 Python sys.version，它總是會包括 patchlevel（預設🄵 '0'）。

platform.**release**()

回傳系統的釋出版本，例如 '2.2.0' 或 'NT'，如果該值無法確定則將回傳一個空字串。

platform.**system**()

回傳系統/OS 的名稱，例如 'Linux'、'Darwin'、'Java'、'Windows'。如果該值無法確定則回傳一個空字串。

On iOS and Android, this returns the user-facing OS name (i.e, 'iOS', 'iPadOS' or 'Android'). To obtain the kernel name ('Darwin' or 'Linux'), use *os.uname()*.

platform.**system_alias**(*system*, *release*, *version*)

回傳做🄵某些系統所使用的常見行銷名稱之🄵名的 (system, release, version)。它還會在可能導致混淆的情🄵下對資訊進行一些重新排序。

platform.**version**()

回傳系統的釋出版本資訊，例如 '#3 on degas'。如果該值無法確定則將回傳一個空字串。

On iOS and Android, this is the user-facing OS version. To obtain the Darwin or Linux kernel version, use *os.uname()*.

platform.**uname**()

具有高可🄵性 (portable) 的 uname 介面。回傳包含六個屬性的*namedtuple()*：*system*、*node*、*release*、*version*、*machine* 和*processor*。

processor 會延遲解析，有需求時才會解析

注意：前兩個屬性名稱與*os.uname()* 提供的名稱不同，它們分🄵命名🄵 sysname 和 nodename。

無法確定的條目會被設🄵 ''。

在 3.3 版的變更: 將結果從 tuple 改🄵*namedtuple()*。

在 3.9 版的變更: *processor* 會延遲解析，🄵非立即解析。

platform.**invalidate_caches**()

Clear out the internal cache of information, such as the *uname()*. This is typically useful when the platform's *node()* is changed by an external process and one needs to retrieve the updated value.

在 3.14 版被加入.

16.7.2 Java 平台

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Jython 的版本介面。

回傳一個 tuple (release, vendor, vminfo, osinfo)，其中 *vminfo* 🄵 tuple (vm_name, vm_release, vm_vendor) 而 *osinfo* 🄵 tuple (os_name, os_version, os_arch)。無法確定的值將被設🄵由參數所給定的預設值（預設均🄵 ''）。

Deprecated since version 3.13, will be removed in version 3.15: It was largely untested, had a confusing API, and was only useful for Jython support.

16.7.3 Windows 平台

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

從 Windows 登🄵檔 (Window Registry) 取得額外的版本資訊🄵回傳一個 tuple (release, version, csd, ptype)，它代表 OS 發行版、版本號、CSD 級🄵 (service pack) 和 OS 類型（多個/單個處理器）。

一點提示：*ptype* 在單個處理器的 NT 機器上🄵 'Uniprocessor Free'，而在多個處理器的機器上🄵 'Multiprocessor Free'。'Free' 是指該 OS 版本🄵有除錯程式。它也可能以 'Checked' 表示，代表該 OS 版本使用了除錯程式，即檢查引數、範圍等的程式。

platform.**win32_edition**()

回傳一個代表目前 Windows 版本的字串。可能的值包括但不限於 'Enterprise'、'IoTUAP'、'ServerStandard' 和 'nanoserver'。

在 3.8 版被加入.

platform.**win32_is_iot**()

如果*win32_edition()* 回傳的 Windows 版本被識🄵🄵 IoT 版則回傳 True。

在 3.8 版被加入.

16.7.4 macOS 平台

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

取得 Mac OS 版本資訊🄵將其回傳🄵 tuple `(release, versioninfo, machine)`，其中 *versioninfo* 是一個 tuple `(version, dev_stage, non_release_version)`。

無法確定的條目會被設🄵 `''`。所有 tuple 條目均🄵字串。

16.7.5 iOS 平台

platform.**ios_ver**(*system=''*, *release=''*, *model=''*, *is_simulator=False*)

Get iOS version information and return it as a *namedtuple()* with the following attributes:

- `system` 是 OS 名稱；可能是 `'iOS'` 或 `'iPadOS'`。
- `release` is the iOS version number as a string (e.g., `'17.2'`).
- `model` is the device model identifier; this will be a string like `'iPhone13,2'` for a physical device, or `'iPhone'` on a simulator.
- `is_simulator` is a boolean describing if the app is running on a simulator or a physical device.

無法確定的條目會被設定🄵以參數給定的預設值。

16.7.6 Unix 平台

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

嘗試確認可執行檔案（預設🄵 Python 直譯器）所連結到的 libc 版本。回傳一個字串 tuple `(lib, version)`，當查詢失敗時其預設值將被設🄵給定的參數值。

請注意，此函式對於不同 libc 版本如何🄵可執行檔案新增符號的方式有深層的關聯，可能僅適用於以 **gcc** 編譯出來的可執行檔案。

檔案會以 *chunksize* 位元組大小的分塊 (chunk) 來讀取和掃描。

16.7.7 Linux 平台

platform.**freedesktop_os_release**()

從 `os-release` 檔案取得作業系統標識，🄵將其作🄵一個字典回傳。`os-release` 檔案🄵 freedesktop.org 標准、🄵在大多數 Linux 發行版上可用。一個重要的例外是 Android 和基於 Android 的發行版。

當 `/etc/os-release` 與 `/usr/lib/os-release` 均無法被讀取時將引發*OSError* 或其子類🄵。

成功時，該函式將回傳一個字典，其中鍵和值均🄵字串。值當中的特殊字元例如 `"` 和 `$` 會被移除引號 (unquoted)。欄位 `NAME`、`ID` 和 `PRETTY_NAME` 總會按照標准來定義。所有其他欄位都是可選的。根據不同廠商可能會包括額外的欄位。

請注意 `NAME`、`VERSION` 和 `VARIANT` 等欄位是適用於向使用者展示的字串。程式應當使用 `ID`、`ID_LIKE`、`VERSION_ID` 或 `VARIANT_ID` 等欄位來標識 Linux 發行版。

範例：

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids are space separated and ordered by precedence
        ids.extend(info["ID_LIKE"].split())
    return ids
```

在 3.10 版被加入.

16.7.8 Android 平台

platform.**android_ver**(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

Get Android device information. Returns a *namedtuple()* with the following attributes. Values which cannot be determined are set to the defaults given as parameters.

- release - Android version, as a string (e.g. "14").
- api_level - API level of the running device, as an integer (e.g. 34 for Android 14). To get the API level which Python was built against, see *sys.getandroidapilevel()*.
- manufacturer - Manufacturer name.
- model - Model name –typically the marketing name or model number.
- device - Device name –typically the model number or a codename.
- is_emulator - True if the device is an emulator; False if it's a physical device.

Google maintains a list of known model and device names.

在 3.13 版被加入.

16.7.9 命令列用法

platform can also be invoked directly using the -m switch of the interpreter:

```
python -m platform [--terse] [--nonaliased] [{nonaliased,terse} ...]
```

可接受以下選項:

--terse

Print terse information about the platform. This is equivalent to calling *platform.platform()* with the *terse* argument set to True.

--nonaliased

Print platform information without system/OS name aliasing. This is equivalent to calling *platform.platform()* with the *aliased* argument set to True.

You can also pass one or more positional arguments (terse, nonaliased) to explicitly control the output format. These behave similarly to their corresponding options.

16.8 errno --- 標準 errno 系統符號

This module makes available standard errno system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from linux/include/errno.h, which should be all-inclusive.

errno.**errorcode**

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, errno.errorcode[errno.EPERM] maps to 'EPERM'.

To translate a numeric error code to an error message, use *os.strerror()*.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as errno.errorcode.keys(). Symbols available can include:

errno.**EPERM**

Operation not permitted. This error is mapped to the exception *PermissionError*.

errno.**ENOENT**

No such file or directory. This error is mapped to the exception *FileNotFoundError*.

errno.**ESRCH**
No such process. This error is mapped to the exception *ProcessLookupError*.

errno.**EINTR**
Interrupted system call. This error is mapped to the exception *InterruptedError*.

errno.**EIO**
I/O error

errno.**ENXIO**
No such device or address

errno.**E2BIG**
Arg list too long

errno.**ENOEXEC**
Exec format error

errno.**EBADF**
Bad file number

errno.**ECHILD**
No child processes. This error is mapped to the exception *ChildProcessError*.

errno.**EAGAIN**
Try again. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMEM**
Out of memory

errno.**EACCES**
Permission denied. This error is mapped to the exception *PermissionError*.

errno.**EFAULT**
Bad address

errno.**ENOTBLK**
Block device required

errno.**EBUSY**
Device or resource busy

errno.**EEXIST**
File exists. This error is mapped to the exception *FileExistsError*.

errno.**EXDEV**
Cross-device link

errno.**ENODEV**
No such device

errno.**ENOTDIR**
Not a directory. This error is mapped to the exception *NotADirectoryError*.

errno.**EISDIR**
Is a directory. This error is mapped to the exception *IsADirectoryError*.

errno.**EINVAL**
Invalid argument

errno.**ENFILE**
File table overflow

errno.**EMFILE**
: Too many open files

errno.**ENOTTY**
: Not a typewriter

errno.**ETXTBSY**
: Text file busy

errno.**EFBIG**
: File too large

errno.**ENOSPC**
: No space left on device

errno.**ESPIPE**
: Illegal seek

errno.**EROFS**
: Read-only file system

errno.**EMLINK**
: Too many links

errno.**EPIPE**
: Broken pipe. This error is mapped to the exception *BrokenPipeError*.

errno.**EDOM**
: Math argument out of domain of func

errno.**ERANGE**
: Math result not representable

errno.**EDEADLK**
: Resource deadlock would occur

errno.**ENAMETOOLONG**
: File name too long

errno.**ENOLCK**
: No record locks available

errno.**ENOSYS**
: Function not implemented

errno.**ENOTEMPTY**
: Directory not empty

errno.**ELOOP**
: Too many symbolic links encountered

errno.**EWOULDBLOCK**
: Operation would block. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMSG**
: No message of desired type

errno.**EIDRM**
: Identifier removed

errno.**ECHRNG**
: Channel number out of range

errno.**EL2NSYNC**
Level 2 not synchronized

errno.**EL3HLT**
Level 3 halted

errno.**EL3RST**
Level 3 reset

errno.**ELNRNG**
Link number out of range

errno.**EUNATCH**
Protocol driver not attached

errno.**ENOCSI**
No CSI structure available

errno.**EL2HLT**
Level 2 halted

errno.**EBADE**
Invalid exchange

errno.**EBADR**
Invalid request descriptor

errno.**EXFULL**
Exchange full

errno.**ENOANO**
No anode

errno.**EBADRQC**
Invalid request code

errno.**EBADSLT**
Invalid slot

errno.**EDEADLOCK**
File locking deadlock error

errno.**EBFONT**
Bad font file format

errno.**ENOSTR**
Device not a stream

errno.**ENODATA**
No data available

errno.**ETIME**
Timer expired

errno.**ENOSR**
Out of streams resources

errno.**ENONET**
Machine is not on the network

errno.**ENOPKG**
Package not installed

errno.**EREMOTE**
Object is remote

errno.**ENOLINK**
Link has been severed

errno.**EADV**
Advertise error

errno.**ESRMNT**
Srmount error

errno.**ECOMM**
Communication error on send

errno.**EPROTO**
Protocol error

errno.**EMULTIHOP**
Multihop attempted

errno.**EDOTDOT**
RFS specific error

errno.**EBADMSG**
Not a data message

errno.**EOVERFLOW**
Value too large for defined data type

errno.**ENOTUNIQ**
Name not unique on network

errno.**EBADFD**
File descriptor in bad state

errno.**EREMCHG**
Remote address changed

errno.**ELIBACC**
Can not access a needed shared library

errno.**ELIBBAD**
Accessing a corrupted shared library

errno.**ELIBSCN**
.lib section in a.out corrupted

errno.**ELIBMAX**
Attempting to link in too many shared libraries

errno.**ELIBEXEC**
Cannot exec a shared library directly

errno.**EILSEQ**
Illegal byte sequence

errno.**ERESTART**
Interrupted system call should be restarted

errno.**ESTRPIPE**
Streams pipe error

errno.**EUSERS**
使用者數量過多

errno.**ENOTSOCK**
Socket operation on non-socket

errno.**EDESTADDRREQ**
Destination address required

errno.**EMSGSIZE**
Message too long

errno.**EPROTOTYPE**
Protocol wrong type for socket

errno.**ENOPROTOOPT**
Protocol not available

errno.**EPROTONOSUPPORT**
Protocol not supported

errno.**ESOCKTNOSUPPORT**
Socket type not supported

errno.**EOPNOTSUPP**
Operation not supported on transport endpoint

errno.**ENOTSUP**
Operation not supported
在 3.2 版被加入.

errno.**EPFNOSUPPORT**
Protocol family not supported

errno.**EAFNOSUPPORT**
Address family not supported by protocol

errno.**EADDRINUSE**
Address already in use

errno.**EADDRNOTAVAIL**
Cannot assign requested address

errno.**ENETDOWN**
Network is down

errno.**ENETUNREACH**
Network is unreachable

errno.**ENETRESET**
Network dropped connection because of reset

errno.**ECONNABORTED**
Software caused connection abort. This error is mapped to the exception *ConnectionAbortedError*.

errno.**ECONNRESET**
Connection reset by peer. This error is mapped to the exception *ConnectionResetError*.

errno.**ENOBUFS**
No buffer space available

errno.**EISCONN**
Transport endpoint is already connected

errno.**ENOTCONN**
Transport endpoint is not connected

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown. This error is mapped to the exception *BrokenPipeError*.

errno.**ETOOMANYREFS**
Too many references: cannot splice

errno.**ETIMEDOUT**
Connection timed out. This error is mapped to the exception *TimeoutError*.

errno.**ECONNREFUSED**
Connection refused. This error is mapped to the exception *ConnectionRefusedError*.

errno.**EHOSTDOWN**
Host is down

errno.**EHOSTUNREACH**
No route to host

errno.**EHWPOISON**
Memory page has hardware error.

在 3.14 版被加入.

errno.**EALREADY**
Operation already in progress. This error is mapped to the exception *BlockingIOError*.

errno.**EINPROGRESS**
Operation now in progress. This error is mapped to the exception *BlockingIOError*.

errno.**ESTALE**
Stale NFS file handle

errno.**EUCLEAN**
Structure needs cleaning

errno.**ENOTNAM**
Not a XENIX named type file

errno.**ENAVAIL**
No XENIX semaphores available

errno.**EISNAM**
Is a named type file

errno.**EREMOTEIO**
Remote I/O error

errno.**EDQUOT**
Quota exceeded

errno.**EQFULL**
Interface output queue is full

在 3.11 版被加入.

errno.**ENOMEDIUM**
No medium found

errno.**EMEDIUMTYPE**
Wrong medium type

errno.**ENOKEY**
Required key not available

errno.**EKEYEXPIRED**
Key has expired

errno.**EKEYREVOKED**
Key has been revoked

errno.**EKEYREJECTED**
Key was rejected by service

errno.**ERFKILL**
Operation not possible due to RF-kill

errno.**ELOCKUNMAPPED**
Locked lock was unmapped

errno.**ENOTACTIVE**
Facility is not active

errno.**EAUTH**
Authentication error

在 3.2 版被加入.

errno.**EBADARCH**
Bad CPU type in executable

在 3.2 版被加入.

errno.**EBADEXEC**
Bad executable (or shared library)

在 3.2 版被加入.

errno.**EBADMACHO**
Malformed Mach-o file

在 3.2 版被加入.

errno.**EDEVERR**
裝置錯誤

在 3.2 版被加入.

errno.**EFTYPE**
不適當的檔案類型或格式

在 3.2 版被加入.

errno.**ENEEDAUTH**
需要驗證器

在 3.2 版被加入.

errno.**ENOATTR**
找不到屬性

在 3.2 版被加入.

errno.**ENOPOLICY**
Policy not found
在 3.2 版被加入.

errno.**EPROCLIM**
行程數量過多
在 3.2 版被加入.

errno.**EPROCUNAVAIL**
Bad procedure for program
在 3.2 版被加入.

errno.**EPROGMISMATCH**
Program version wrong
在 3.2 版被加入.

errno.**EPROGUNAVAIL**
RPC prog. not avail
在 3.2 版被加入.

errno.**EPWROFF**
裝置電源關閉
在 3.2 版被加入.

errno.**EBADRPC**
RPC 結構不正確
在 3.2 版被加入.

errno.**ERPCMISMATCH**
RPC 版本錯誤
在 3.2 版被加入.

errno.**ESHLIBVERS**
共享函式庫版本不匹配
在 3.2 版被加入.

errno.**ENOTCAPABLE**
Capabilities insufficient. This error is mapped to the exception *PermissionError*.
可用性: WASI, FreeBSD
在 3.11.1 版被加入.

errno.**ECANCELED**
操作已取消
在 3.2 版被加入.

errno.**EOWNERDEAD**
擁有者過世
在 3.2 版被加入.

errno.**ENOTRECOVERABLE**
狀態無法恢復
在 3.2 版被加入.

16.9 ctypes --- 用於 Python 的外部函式庫

原始碼： Lib/ctypes

---

ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

16.9.1 ctypes 教學

Note: The code samples in this tutorial use *doctest* to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or macOS, they contain doctest directives in comments.

Note: Some code samples reference the ctypes *c_int* type. On platforms where `sizeof(long) == sizeof(int)` it is an alias to *c_long*. So, you should not be confused if *c_long* is printed if you would expect *c_int* --- they are actually the same type.

Loading dynamic link libraries

ctypes exports the *cdll*, and on Windows *windll* and *oledll* objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise an *OSError* exception when the function call fails.

在 3.3 版的變更: Windows errors used to raise *WindowsError*, which is now an alias of *OSError*.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the `cdecl` calling convention:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows appends the usual `.dll` file suffix automatically.

> **備🄵**
>
> Accessing the standard C library through `cdll.msvcrt` will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the `msvcrt` module.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the *LoadLibrary()* method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with a `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI 版本 */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE 版本 */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like `"??2@YAPAXI@Z"`. In this case you have to use *`getattr()`* to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

呼叫函式

You can call these functions like any other Python callable. This example uses the `rand()` function, which takes no arguments and returns a pseudo-random integer:

```
>>> print(libc.rand())
1804289383
```

On Windows, you can call the `GetModuleHandleA()` function, which returns a win32 module handle (passing `None` as single argument to call it with a `NULL` pointer):

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

ValueError is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, *ctypes* uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with *ctypes*, so you should be careful anyway. The *faulthandler* module can be helpful in debugging crashes (e.g. from segmentation faults produced by erroneous C library calls).

`None`, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (`char*` or `wchar_t*`). Python integers are passed as the platform's default C `int` type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about *ctypes* data types.

Fundamental data types

ctypes defines a number of primitive C compatible data types:

ctypes 型🄵	C 型🄵	Python 型🄵
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_int8	int8_t	int
c_int16	int16_t	int
c_int32	int32_t	int
c_int64	int64_t	int
c_uint	unsigned int	int

繼續下一頁

表格 1 – 繼續上一頁

ctypes 型🄵	C 型🄵	Python 型🄵
c_uint8	uint8_t	int
c_uint16	uint16_t	int
c_uint32	uint32_t	int
c_uint64	uint64_t	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 或 long long	int
c_ulonglong	unsigned __int64 或 unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t 或 Py_ssize_t	int
c_time_t	time_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (NUL terminated)	位元組物件或 None
c_wchar_p	wchar_t* (NUL terminated)	字串或 None
c_void_p	void*	int or None

(1) The constructor accepts any object with a truth value.

Additionally, if IEC 60559 compatible complex arithmetic (Annex G) is supported in both C and libffi, the following complex types are available:

ctypes 型🄵	C 型🄵	Python 型🄵
c_float_complex	float complex	complex
c_double_complex	double complex	complex
c_longdouble_complex	long double complex	complex

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p*, and *c_void_p* changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python string objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
```

(繼續下一頁)

(繼續上一頁)

```
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # 記憶體位置已改變
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # 第一個物件未改變
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a *create_string_buffer()* function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL terminated␣
↪string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The *create_string_buffer()* function replaces the old `c_buffer()` function (which is still available as an alias). To create a mutable memory block containing unicode characters of the C type `wchar_t`, use the *create_unicode_buffer()* function.

Calling functions, continued

Note that printf prints to the real standard output channel, *not* to *sys.stdout*, so these examples will only work at the console prompt, not from within *IDLE* or *PythonWin*:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding *ctypes* type, so that they can be converted to the required C data type:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the *argtypes* attribute for the regular, non-variadic, function arguments:

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does not inhibit portability it is advised to always specify *argtypes* for all variadic functions.

Calling functions with your own custom data types

You can also customize *ctypes* argument conversion to allow instances of your own classes be used as function arguments. *ctypes* looks for an `_as_parameter_` attribute and uses this as the function argument. The attribute must be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the `_as_parameter_` instance variable, you could define a *property* which makes the attribute available on request.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the *argtypes* attribute.

argtypes must be a sequence of C data types (the `printf()` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(繼續下一頁)

(繼續上一頁)

```
ctypes.ArgumentError: argument 2: TypeError: 'int' object cannot be interpreted as ctypes.c_
↪char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a *from_param()* class method for them to be able to use them in the *argtypes* sequence. The *from_param()* class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute.

Return types

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the *restype* attribute of the function object.

The C prototype of `time()` is `time_t time(time_t *)`. Because `time_t` might be of a different type than the default return type `int`, you should specify the `restype` attribute:

```
>>> libc.time.restype = c_time_t
```

The argument types can be specified using *argtypes*:

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

To call the function with a `NULL` pointer as first argument, use `None`:

```
>>> print(libc.time(None))
1150640792
```

Here is a more advanced example, it uses the `strchr()` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the *ord("x")* calls above, you can set the *argtypes* attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character bytes, bytearray or integer␣
↪expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
```

(繼續下一頁)

(繼續上一頁)

```
b'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the *restype* attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls *GetLastError()* to retrieve it.

Please note that a much more powerful error checking mechanism is available through the *errcheck* attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

ctypes exports the *byref()* function which is used to pass parameters by reference. The same effect can be achieved with the *pointer()* function, although *pointer()* does a lot more work since it constructs a real pointer object, so it is faster to use *byref()* if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the *Structure* and *Union* base classes which are defined in the *ctypes* module. Each subclass must define a *_fields_* attribute. `_fields_` must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a *ctypes* type like *c_int*, or any other derived *ctypes* type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named *x* and *y*, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field *descriptor*s can be retrieved from the *class*, they are useful for debugging because they can provide useful information. See *CField*:

```
>>> POINT.x
<ctypes.CField 'x' type=c_int, ofs=0, size=4>
>>> POINT.y
<ctypes.CField 'y' type=c_int, ofs=4, size=4>
>>>
```

警告

ctypes does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union layout, alignment and byte order

By default, Structure and Union fields are laid out in the same way the C compiler does it. It is possible to override this behavior entirely by specifying a *_layout_* class attribute in the subclass definition; see the attribute documentation for details.

It is possible to specify the maximum alignment for the fields by setting the *_pack_* class attribute to a positive integer. This matches what `#pragma pack(n)` does in MSVC.

It is also possible to set a minimum alignment for how the subclass itself is packed in the same way `#pragma align(n)` works in MSVC. This can be achieved by specifying a *_align_* class attribute in the subclass definition.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion*, and *LittleEndianUnion* base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the *_fields_* tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<ctypes.CField 'first_16' type=c_int, ofs=0, bit_size=16, bit_offset=0>
>>> print(Int.second_16)
<ctypes.CField 'second_16' type=c_int, ofs=0, bit_size=16, bit_offset=16>
```

It is important to note that bit field allocation and layout in memory are not defined as a C standard; their implementation is compiler-specific. By default, Python will attempt to match the behavior of a "native" compiler for the current platform. See the *_layout_* attribute for details on the default behavior and how to change it.

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of `0 0` lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

指標

Pointer instances are created by calling the *pointer()* function on a *ctypes* type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a *contents* attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that *ctypes* does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another *c_int* instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the *pointer()* function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the *POINTER()* function, which accepts any *ctypes* type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Thread safety without the GIL

From Python 3.13 onward, the *GIL* can be disabled on *free threaded* builds. In ctypes, reads and writes to a single object concurrently is safe, but not across multiple objects:

```
>>> number = c_int(42)
>>> pointer_a = pointer(number)
>>> pointer_b = pointer(number)
```

In the above, it's only safe for one object to read and write to the address at once if the GIL is disabled. So, `pointer_a` can be shared and written to across multiple threads, but only if `pointer_b` is not also attempting to do the same. If this is an issue, consider using a *threading.Lock* to synchronize access to memory:

```
>>> import threading
>>> lock = threading.Lock()
>>> # Thread 1
>>> with lock:
...     pointer_a.contents = 24
>>> # Thread 2
>>> with lock:
...     pointer_b.contents = 42
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the *argtypes* list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in *argtypes*, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

To set a POINTER type field to `NULL`, you can assign `None`:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. *ctypes* provides a *cast()* function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or *c_int* arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the *cast()* function is handy.

The *cast()* function can be used to cast a ctypes instance into a pointer to a different ctypes data type. *cast()* takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, *cast()* can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the *_fields_* attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Let's try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

回呼函式

ctypes allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The *CFUNCTYPE()* factory function creates types for callback functions using the `cdecl` calling convention. On Windows, the *WINFUNCTYPE()* factory function creates types for callback functions using the `stdcall` calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

To get started, here is a simple callback that shows the values it gets passed:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

結果🄵:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Now we can actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

As we can easily check, our array is sorted now:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

The function factories can be used as decorator factories, so we may as well write:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

備🄵

Make sure you keep references to *CFUNCTYPE()* objects as long as they are used from C code. *ctypes* doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with *threading.local* will *not* survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_Version`, Python runtime version number encoded in a single constant integer.

ctypes can access values like this with the *in_dll()* class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi, "Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the docs for that value:

> This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with *ctypes*:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # 函式指標
...                ]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "_PyImport_FrozenBootstrap")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the `NULL` entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative `size` member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edges in *ctypes* where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` 's buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave differently from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

備🄵

Objects instantiated from *c_char_p* can only have their value set to bytes or integers.

Why is it printing `False`? ctypes instances are objects containing a memory block plus some *descriptor*s accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with *ctypes* is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

16.9.2 ctypes reference

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the *find_library()* function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, *find_library()* tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

在 3.6 版的變更: On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.

以下是一些範例:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On macOS and Android, *find_library()* uses the system's standard naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, *find_library()* searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using *find_library()* to locate the library at runtime.

Listing loaded shared libraries

When writing code that relies on code loaded from shared libraries, it can be useful to know which shared libraries have already been loaded into the current process.

The `ctypes.util` module provides the *dllist()* function, which calls the different APIs provided by the various platforms to help determine which shared libraries have already been loaded into the current process.

The exact output of this function will be system dependent. On most platforms, the first entry of this list represents the current process itself, which may be an empty string. For example, on glibc-based Linux, the return may look like:

```
>>> from ctypes.util import dllist
>>> dllist()
```

(繼續下一頁)

(繼續上一頁)

```
['', 'linux-vdso.so.1', '/lib/x86_64-linux-gnu/libm.so.6', '/lib/x86_64-linux-gnu/libc.so.6',
↪ ... ]
```

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

On Windows creating a *CDLL* instance may fail even if the DLL name exists. When a dependent DLL of the loaded DLL is not found, a *OSError* error is raised with the message *"[WinError 126] The specified module could not be found"*. This error message does not contain the name of the missing DLL because the Windows API does not return this information making this error hard to diagnose. To resolve this error and determine which DLL is not found, you need to find the list of dependent DLLs and determine which one is not found using Windows debugging and tracing tools.

在 3.12 版的變更: The *name* parameter can now be a *path-like object*.

也參考

Microsoft DUMPBIN tool -- A tool to find DLL dependents.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific *HRESULT* code. *HRESULT* values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an *OSError* is automatically raised.

可用性: Windows

在 3.3 版的變更: *WindowsError* used to be raised, which is now an alias of *OSError*.

在 3.12 版的變更: The *name* parameter can now be a *path-like object*.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

可用性: Windows

在 3.12 版的變更: The *name* parameter can now be a *path-like object*.

The Python *global interpreter lock* is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like *CDLL* instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

在 3.12 版的變更: The *name* parameter can now be a *path-like object*.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platform's `dlopen()` or `LoadLibrary()` function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen(3)* manpage. On Windows, *mode* is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the system's *errno* variable; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API `LoadLibraryEx` flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load, which avoids issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

在 3.8 版的變更: 新增 *winmode* 參數。

`ctypes.`**`RTLD_GLOBAL`**

Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

`ctypes.`**`RTLD_LOCAL`**

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

`ctypes.`**`DEFAULT_MODE`**

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # 於 Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

`PyDLL.`**`_handle`**

The system handle used to access the library.

`PyDLL.`**`_name`**

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the *LibraryLoader* class, either by calling the *LoadLibrary()* method, or by retrieving the library as attribute of the loader instance.

class ctypes.**LibraryLoader**(*dlltype*)

Class which loads shared libraries. *dlltype* should be one of the *CDLL*, *PyDLL*, *WinDLL*, or *OleDLL* types.

__getattr__() has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.**cdll**

Creates *CDLL* instances.

ctypes.**windll**

Creates *WinDLL* instances.

可用性: Windows

ctypes.**oledll**

Creates *OleDLL* instances.

可用性: Windows

ctypes.**pydll**

Creates *PyDLL* instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.**pythonapi**

An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed to return C int, which is of course not always the truth, so you have to assign the correct restype attribute to use these functions.

Loading a library through any of these objects raises an *auditing event* ctypes.dlopen with string argument name, the name used to load the library.

Accessing a function on a loaded library raises an auditing event ctypes.dlsym with arguments library (the library object) and name (the symbol's name as a string or integer).

In cases when only the library handle is available rather than the object, accessing a function raises an auditing event ctypes.dlsym/handle with arguments handle (the raw library handle) and name.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader.

They are instances of a private local class _FuncPtr (not exposed in ctypes) which inherits from the private *_CFuncPtr* class:

```
>>> import ctypes
>>> lib = ctypes.CDLL(None)
>>> issubclass(lib._FuncPtr, ctypes._CFuncPtr)
True
>>> lib._FuncPtr is ctypes._CFuncPtr
False
```

class ctypes._CFuncPtr

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the *errcheck* attribute.

argtypes

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the *from_param()* class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *c_char_p* item in the *argtypes* tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a *from_param()* method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

On Windows, when a foreign function call raises a system exception (for example, due to an access violation), it will be captured and replaced with a suitable Python exception. Further, an auditing event `ctypes.set_exception` with argument `code` will be raised, allowing an audit hook to replace the exception with its own.

Some ways to invoke foreign function calls as well as some of the functions in this module may raise an auditing event `ctypes.call_function` with arguments `function pointer` and `arguments`.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the `@wrapper` syntax. See 回呼函式 for examples.

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the ctypes private copy of the system `errno` variable is exchanged with the real `errno` value before and after the call; *use_last_error* does the same for the Windows error code.

ctypes.**WINFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

可用性: Windows

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

prototype(*address*)

Returns a foreign function at the specified address which must be an integer.

prototype(*callable*)

Create a C callable function (a callback function) from a Python *callable*.

prototype(*func_spec*[, *paramflags*])

Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

If *iid* is not specified, an `OSError` is raised if the COM method call fails. If *iid* is specified, a `COMError` is raised instead.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

可用性: Windows

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1
: Specifies an input parameter to the function.

2
: Output parameter. The foreign function fills in a value.

4
: Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

The following example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes"), (1,
↪"flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

The `MessageBox` foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the *errcheck* protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the *errcheck* function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Utility functions

ctypes.**addressof**(*obj*)

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

引發一個附帶引數 obj 的稽核事件 ctypes.addressof。

ctypes.**alignment**(*obj_or_type*)

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

ctypes.**byref**(*obj*[, *offset*])

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

ctypes.**CopyComPointer**(*src*, *dst*)

Copies a COM pointer from *src* to *dst* and returns the Windows specific `HRESULT` value.

If *src* is not `NULL`, its `AddRef` method is called, incrementing the reference count.

In contrast, the reference count of *dst* will not be decremented before assigning the new value. Unless *dst* is `NULL`, the caller is responsible for decrementing the reference count by calling its `Release` method when necessary.

可用性: Windows

在 3.14 版被加入.

ctypes.**cast**(*obj*, *type*)

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

ctypes.**create_string_buffer**(*init*, *size=None*)

ctypes.**create_string_buffer**(*size*)

This function creates a mutable character buffer. The returned object is a ctypes array of *c_char*.

If *size* is given (and not `None`), it must be an *int*. It specifies the size of the returned array.

If the *init* argument is given, it must be *bytes*. It is used to initialize the array items. Bytes not initialized this way are set to zero (NUL).

If *size* is not given (or if it is `None`), the buffer is made one element larger than *init*, effectively adding a NUL terminator.

If both arguments are given, *size* must not be less than `len(init)`.

> **警告**
>
> If *size* is equal to `len(init)`, a NUL terminator is not added. Do not treat such a buffer as a C string.

For example:

```
>>> bytes(create_string_buffer(2))
b'\x00\x00'
>>> bytes(create_string_buffer(b'ab'))
b'ab\x00'
>>> bytes(create_string_buffer(b'ab', 2))
b'ab'
>>> bytes(create_string_buffer(b'ab', 4))
b'ab\x00\x00'
>>> bytes(create_string_buffer(b'abcdef', 2))
Traceback (most recent call last):
   ...
ValueError: byte string too long
```

引發一個附帶引數 `init` 與 `size` 的稽核事件 `ctypes.create_string_buffer`。

`ctypes.`**`create_unicode_buffer`**(*init*, *size=None*)

`ctypes.`**`create_unicode_buffer`**(*size*)

This function creates a mutable unicode character buffer. The returned object is a ctypes array of *c_wchar*.

The function takes the same arguments as *create_string_buffer()* except *init* must be a string and *size* counts *c_wchar*.

引發一個附帶引數 `init` 與 `size` 的稽核事件 `ctypes.create_unicode_buffer`。

`ctypes.`**`DllCanUnloadNow`**()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

可用性: Windows

`ctypes.`**`DllGetClassObject`**()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

可用性: Windows

`ctypes.util.`**`find_library`**(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

`ctypes.util.`**`find_msvcrt`**()

Returns the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

可用性: Windows

`ctypes.util.`**`dllist`**()

Try to provide a list of paths of the shared libraries loaded into the current process. These paths are not normalized or processed in any way. The function can raise *OSError* if the underlying platform APIs fail. The exact functionality is system dependent.

On most platforms, the first element of the list represents the current executable file. It may be an empty string.

可用性: Windows, macOS, iOS, glibc, BSD libc, musl

在 3.14 版被加入.

ctypes.**FormatError**([*code*])

Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows API function *GetLastError()*.

可用性: Windows

ctypes.**GetLastError**()

Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

可用性: Windows

ctypes.**get_errno**()

Returns the current value of the ctypes-private copy of the system *errno* variable in the calling thread.

引發一個不附帶引數的稽核事件 `ctypes.get_errno`。

ctypes.**get_last_error**()

Returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

可用性: Windows

引發一個不附帶引數的稽核事件 `ctypes.get_last_error`。

ctypes.**memmove**(*dst*, *src*, *count*)

Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.**memset**(*dst*, *c*, *count*)

Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.**POINTER**(*type*, */*)

Create or return a ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

CPython 實作細節： The resulting pointer type is cached in the `__pointer_type__` attribute of *type*. It is possible to set this attribute before the first call to `POINTER` in order to set a custom pointer type. However, doing this is discouraged: manually creating a suitable pointer type is difficult without relying on implementation details that may change in future Python versions.

ctypes.**pointer**(*obj*, */*)

Create a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.**resize**(*obj*, *size*)

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.**set_errno**(*value*)

Set the current value of the ctypes-private copy of the system *errno* variable in the calling thread to *value* and return the previous value.

引發一個附帶引數 `errno` 的稽核事件 `ctypes.set_errno`。

ctypes.**set_last_error**(*value*)

Sets the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

可用性: Windows

引發一個附帶引數 `error` 的稽核事件 `ctypes.set_last_error`。

ctypes.**sizeof**(*obj_or_type*)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.**string_at**(*ptr*, *size=-1*)

Return the byte string at *void *ptr*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

引發一個附帶引數 `ptr`、`size` 的稽核事件 `ctypes.string_at`。

ctypes.**WinError**(*code=None*, *descr=None*)

Creates an instance of *OSError*. If *code* is not specified, *GetLastError()* is called to determine the error code. If *descr* is not specified, *FormatError()* is called to get a textual description of the error.

可用性: Windows

在 3.3 版的變更: An instance of *WindowsError* used to be created, which is now an alias of *OSError*.

ctypes.**wstring_at**(*ptr*, *size=-1*)

Return the wide-character string at *void *ptr*. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

引發一個附帶引數 `ptr`、`size` 的稽核事件 `ctypes.wstring_at`。

ctypes.**memoryview_at**(*ptr*, *size*, *readonly=False*)

Return a *memoryview* object of length *size* that references memory starting at *void *ptr*.

If *readonly* is true, the returned `memoryview` object can not be used to modify the underlying memory. (Changes made by other means will still be reflected in the returned object.)

This function is similar to *string_at()* with the key difference of not making a copy of the specified memory. It is a semantically equivalent (but more efficient) alternative to `memoryview((c_byte * size).from_address(ptr))`. (While *from_address()* only takes integers, *ptr* can also be given as a *ctypes.POINTER* or a *byref()* object.)

引發一個附帶引數 `address`、`size` 與 `readonly` 的稽核事件 `ctypes.memoryview_at`。

在 3.14 版被加入.

Data types

class ctypes.**_CData**

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the *addressof()* helper function. Another instance variable is exposed as *_objects*; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

引發一個附帶引數 `pointer`、`size`、`offset` 的稽核事件 `ctypes.cdata/buffer`。

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

引發一個附帶引數 `pointer`、`size`、`offset` 的稽核事件 `ctypes.cdata/buffer`。

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

This method, and others that indirectly call this method, raises an *auditing event* `ctypes.cdata` with argument `address`.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's *argtypes* tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common class variables of ctypes data types:

__pointer_type__

The pointer type that was created by calling *POINTER()* for corresponding ctypes data type. If a pointer type was not yet created, the attribute is missing.

在 3.14 版被加入.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Fundamental data types

class ctypes.**_SimpleCData**

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a *restype* of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.**c_byte**

Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**

Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**

Represents the C `char*` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**

Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**

Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

class ctypes.**c_float**

Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_double_complex**

Represents the C `double complex` datatype, if available. The constructor accepts an optional *complex* initializer.

在 3.14 版被加入.

class ctypes.**c_float_complex**

Represents the C `float complex` datatype, if available. The constructor accepts an optional *complex* initializer.

在 3.14 版被加入.

class ctypes.**c_longdouble_complex**

Represents the C `long double complex` datatype, if available. The constructor accepts an optional *complex* initializer.

在 3.14 版被加入.

class ctypes.**c_int**

Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**

Represents the C 8-bit `signed int` datatype. It is an alias for *c_byte*.

class ctypes.**c_int16**

Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**

Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**

Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**

Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**

Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**

Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**

Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**

Represents the C `ssize_t` datatype.

在 3.2 版被加入.

class ctypes.**c_time_t**

Represents the C `time_t` datatype.

在 3.12 版被加入.

class ctypes.**c_ubyte**

Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**

Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**

Represents the C 8-bit `unsigned int` datatype. It is an alias for *c_ubyte*.

class ctypes.**c_uint16**

Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**

Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**

Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**

Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**

Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**

Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**

Represents the C `void*` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**

Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**

Represents the C `wchar_t*` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**

Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.**HRESULT**

Represents a `HRESULT` value, which contains success or error information for a function or method call.

可用性: Windows

class ctypes.**py_object**

Represents the C `PyObject*` datatype. Calling this without an argument creates a `NULL` `PyObject*` pointer.

在 3.14 版的變更: `py_object` is now a *generic type*.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Structured data types

class ctypes.**Union**(**args*, ***kw*)

Abstract base class for unions in native byte order.

Unions share common attributes and behavior with structures; see *`Structure`* documentation for details.

class ctypes.**BigEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *big endian* byte order.

在 3.11 版被加入.

class ctypes.**LittleEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *little endian* byte order.

在 3.11 版被加入.

class ctypes.**BigEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *big endian* byte order.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *little endian* byte order.

Structures and unions with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a *_fields_* class variable. *ctypes* will create *descriptor*s which allow reading and writing the fields by direct attribute accesses. These are the

fields

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like *c_int*, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the *_fields_* class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The _fields_ class variable can only be set once. Later assignments will raise an *AttributeError*.

Additionally, the _fields_ class variable must be defined before the structure or union type is first used: an instance or subclass is created, *sizeof()* is called on it, and so on. Later assignments to _fields_ will raise an *AttributeError*. If _fields_ has not been set before such use, the structure or union will have no own fields, as if _fields_ was empty.

Sub-subclasses of structure types inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. *_pack_* must already be defined when *_fields_* is assigned, otherwise it will have no effect. Setting this attribute to 0 is the same as not setting it at all.

This is only implemented for the MSVC-compatible memory layout.

Deprecated since version 3.14, will be removed in version 3.19: For historical reasons, if _pack_ is non-zero, the MSVC-compatible layout will be used by default. On non-Windows platforms, this default is deprecated and is slated to become an error in Python 3.19. If it is intended, set *_layout_* to 'ms' explicitly.

align

An optional small integer that allows overriding the alignment of the structure when being packed or unpacked to/from memory. Setting this attribute to 0 is the same as not setting it at all.

在 3.13 版被加入.

layout

An optional string naming the struct/union layout. It can currently be set to:

- "ms": the layout used by the Microsoft compiler (MSVC). On GCC and Clang, this layout can be selected with __attribute__((ms_struct)).
- "gcc-sysv": the layout used by GCC with the System V or "SysV-like" data model, as used on Linux and macOS. With this layout, *_pack_* must be unset or zero.

If not set explicitly, ctypes will use a default that matches the platform conventions. This default may change in future Python releases (for example, when a new platform gains official support, or when a difference between similar platforms is found). Currently the default will be:

- On Windows: `"ms"`
- When *_pack_* is specified: `"ms"`. (This is deprecated; see *_pack_* documentation.)
- Otherwise: `"gcc-sysv"`

`_layout_` must already be defined when *_fields_* is assigned, otherwise it will have no effect.

在 3.14 版被加入.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. *_anonymous_* must be already defined when *_fields_* is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in *_fields_*. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize *_fields_* with the same name, or create new attributes for names not present in *_fields_*.

class ctypes.**CField**(**args*, ***kw*)

Descriptor for fields of a *Structure* and *Union*. For example:

```
>>> class Color(Structure):
...     _fields_ = (
...         ('red', c_uint8),
...         ('green', c_uint8),
...         ('blue', c_uint8),
...         ('intense', c_bool, 1),
...         ('blinking', c_bool, 1),
...     )
...
>>> Color.red
<ctypes.CField 'red' type=c_ubyte, ofs=0, size=1>
>>> Color.green.type
```

(繼續下一頁)

(繼續上一頁)

```
<class 'ctypes.c_ubyte'>
>>> Color.blue.byte_offset
2
>>> Color.intense
<ctypes.CField 'intense' type=c_bool, ofs=3, bit_size=1, bit_offset=0>
>>> Color.blinking.bit_offset
1
```

All attributes are read-only.

`CField` objects are created via *`_fields_`*; do not instantiate the class directly.

在 3.14 版被加入: Previously, descriptors only had `offset` and `size` attributes and a readable string representation; the `CField` class was not available directly.

name

Name of the field, as a string.

type

Type of the field, as a *ctypes class*.

offset

byte_offset

Offset of the field, in bytes.

For bitfields, this is the offset of the underlying byte-aligned *storage unit*; see *`bit_offset`*.

byte_size

Size of the field, in bytes.

For bitfields, this is the size of the underlying *storage unit*. Typically, it has the same size as the bitfield's type.

size

For non-bitfields, equivalent to *`byte_size`*.

For bitfields, this contains a backwards-compatible bit-packed value that combines *`bit_size`* and *`bit_offset`*. Prefer using the explicit attributes instead.

is_bitfield

True if this is a bitfield.

bit_offset

bit_size

The location of a bitfield within its *storage unit*, that is, within *`byte_size`* bytes of memory starting at *`byte_offset`*.

To get the field's value, read the storage unit as an integer, shift left by `bit_offset` and take the `bit_size` least significant bits.

For non-bitfields, `bit_offset` is zero and `bit_size` is equal to `byte_size * 8`.

is_anonymous

True if this field is anonymous, that is, it contains nested sub-fields that should be merged into a containing structure or union.

Arrays and pointers

class ctypes.**Array**(**args*)

Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any *`ctypes`* data type with a non-negative integer. Alternatively, you can subclass this type and define *`_length_`* and *`_type_`* class variables.

Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an *IndexError*. Will be returned by *len()*.

type

Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

ctypes.**ARRAY**(*type*, *length*)

Create an array. Equivalent to `type * length`, where *type* is a *ctypes* data type and *length* an integer.

This function is *soft deprecated* in favor of multiplication. There are no plans to remove it.

class ctypes.**_Pointer**

Private, abstract base class for pointers.

Concrete pointer types are created by calling *POINTER()* with the type that will be pointed to; this is done automatically by *pointer()*.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so *len()* will raise *TypeError*. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type

Specifies the type pointed to.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

Exceptions

exception ctypes.**ArgumentError**

This exception is raised when a foreign function call cannot convert one of the passed arguments.

exception ctypes.**COMError**(*hresult*, *text*, *details*)

This exception is raised when a COM method call failed.

hresult

The integer value representing the error code.

text

The error message.

details

The 5-tuple `(descr, source, helpfile, helpcontext, progid)`.

descr is the textual description. *source* is the language-dependent `ProgID` for the class or application that raised the error. *helpfile* is the path of the help file. *helpcontext* is the help context identifier. *progid* is the `ProgID` of the interface that defined the error.

可用性: Windows

在 3.14 版被加入.

CHAPTER 17

命令列介面函式庫

本章節所描述的模組協助實作應用程式的命令列與終端介面。

以下🄵概覽：

17.1 `argparse` --- 命令列選項、引數和子命令的剖析器

在 3.2 版被加入.

原始碼： Lib/argparse.py

備🄵

While *`argparse`* is the default recommended standard library module for implementing basic command line applications, authors with more exacting requirements for exactly how their command line applications behave may find it doesn't provide the necessary level of control. Refer to *選擇一個命令列參數剖析函式庫* for alternatives to consider when `argparse` doesn't support behaviors that the application requires (such as entirely disabling support for interspersed options and positional arguments, or accepting option parameter values that start with `-` even when they correspond to another defined option).

教學

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the *argparse tutorial*.

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of *`sys.argv`*. The `argparse` module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

The `argparse` module's support for command-line interfaces is built around an instance of *`argparse.ArgumentParser`*. It is a container for argument specifications and has options that apply to the parser as whole:

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

The *ArgumentParser.add_argument()* method attaches individual argument specifications to the parser. It supports positional arguments, options that accept values, and on/off flags:

```
parser.add_argument('filename')           # 位置引數
parser.add_argument('-c', '--count')      # 接收一個值的選項
parser.add_argument('-v', '--verbose',
                    action='store_true')  # 開關旗標
```

The *ArgumentParser.parse_args()* method runs the parser and places the extracted data in a *argparse.Namespace* object:

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

備🄵

If you're looking for a guide about how to upgrade *optparse* code to `argparse`, see *Upgrading Optparse Code*.

17.1.1 ArgumentParser 物件

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*, ***, *suggest_on_error=False*, *color=True*)

Create a new *ArgumentParser* object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- *prog* - 程式的名稱（預設值：從 `__main__` 模組屬性和 `sys.argv[0]` 🄵生）
- *usage* - 描述程式用法的字串（預設值：從新增到剖析器的引數🄵生）
- *description* - 引數🄵明之前要顯示的文字（預設值：無文字）
- *epilog* - 引數🄵明之後要顯示的文字（預設值：無文字）
- *parents* - 一個*ArgumentParser* 物件的串列，其引數也應該被包含
- *formatter_class* - 用於自訂🄵明輸出的類🄵
- *prefix_chars* - 前綴可選引數的字元集合（預設值：'-'）
- *fromfile_prefix_chars* - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- *argument_default* - The global default value for arguments (default: `None`)
- *conflict_handler* - The strategy for resolving conflicting optionals (usually unnecessary)
- *add_help* - Add a `-h/--help` option to the parser (default: `True`)
- *allow_abbrev* - Allows long options to be abbreviated if the abbreviation is unambiguous (default: `True`)
- *exit_on_error* - Determines whether or not `ArgumentParser` exits with error info when an error occurs. (default: `True`)
- *suggest_on_error* - Enables suggestions for mistyped argument choices and subparser names (default: `False`)

- *color* - Allow color output (default: `True`)

在 3.5 版的變更: 新增 *allow_abbrev* 參數。

在 3.8 版的變更: In previous versions, *allow_abbrev* also disabled grouping of short flags such as `-vv` to mean `-v -v`.

在 3.9 版的變更: 新增 *exit_on_error* 參數。

在 3.14 版的變更: 新增 *suggest_on_error* 和 *color* 參數。

The following sections describe how each of these are used.

prog

By default, *ArgumentParser* calculates the name of the program to display in help messages depending on the way the Python interpreter was run:

- The *base name* of `sys.argv[0]` if a file was passed as argument.
- The Python interpreter name followed by `sys.argv[0]` if a directory or a zipfile was passed as argument.
- The Python interpreter name followed by `-m` followed by the module or package name if the `-m` option was used.

This default is almost always desirable because it will make the help messages match the string that was used to invoke the program on the command line. However, to change this default behavior, another value can be supplied using the `prog=` argument to *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

options:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]`, from the `__main__` module attributes or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

在 3.14 版的變更: The default `prog` value now reflects how `__main__` was actually executed, rather than always being `os.path.basename(sys.argv[0])`.

usage

By default, *ArgumentParser* calculates the usage message from the arguments it contains. The default message can be overridden with the `usage=` keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help
```

(繼續下一頁)

(繼續上一頁)

```
options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The `%(prog)s` format specifier is available to fill in the program name in your usage messages.

When a custom usage message is specified for the main parser, you may also want to consider passing the `prog` argument to *`add_subparsers()`* or the `prog` and the `usage` arguments to `add_parser()`, to ensure consistent command prefixes and usage information across subparsers.

description

Most calls to the *`ArgumentParser`* constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments.

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the *formatter_class* argument.

epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to *`ArgumentParser`*:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

As with the *description* argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the *formatter_class* argument to *`ArgumentParser`*.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to *`ArgumentParser`* can be used. The `parents=` argument takes a list of *`ArgumentParser`* objects, collects all the positional and optional actions from them, and adds these actions to the *`ArgumentParser`* object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify `add_help=False`. Otherwise, the *`ArgumentParser`* will see two `-h/--help` options (one in the parent and one in the child) and raise an error.

> **備🄵**
>
> You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

formatter_class

`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

class argparse.**RawDescriptionHelpFormatter**

class argparse.**RawTextHelpFormatter**

class argparse.**ArgumentDefaultsHelpFormatter**

class argparse.**MetavarTypeHelpFormatter**

`RawDescriptionHelpFormatter` and *`RawTextHelpFormatter`* give more control over how textual descriptions are displayed. By default, *`ArgumentParser`* objects line-wrap the *description* and *epilog* texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passing *`RawDescriptionHelpFormatter`* as `formatter_class=` indicates that *description* and *epilog* are already correctly formatted and should not be line-wrapped:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
```

(繼續下一頁)

(繼續上一頁)

```
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple newlines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

ArgumentDefaultsHelpFormatter automatically adds information about default values to each of the argument help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter uses the name of the *type* argument for each argument as the display name for its values (rather than using the *dest* as the regular formatter does):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

Most command-line options will use `-` as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the *ArgumentParser* constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include `-` will cause `-f/--foo` options to be disallowed.

fromfile_prefix_chars

Sometimes, when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *`ArgumentParser`* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w', encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must be one per line by default (but see also *`convert_arg_line_to_args()`*) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

備🄵

Empty lines are treated as empty strings (`''`), which are allowed as values but not as arguments. Empty lines that are read as arguments will result in an ”unrecognized arguments” error.

`ArgumentParser` uses *filesystem encoding and error handler* to read the file containing arguments.

The `fromfile_prefix_chars=` argument defaults to `None`, meaning that arguments will never be treated as file references.

在 3.12 版的變更: *`ArgumentParser`* changed encoding and errors to read arguments files from default (e.g. *`locale.getpreferredencoding(False)`* and `"strict"`) to the *filesystem encoding and error handler*. Arguments file should be encoded in UTF-8 instead of ANSI Codepage on Windows.

argument_default

Generally, argument defaults are specified either by passing a default to *`add_argument()`* or by calling the *`set_defaults()`* methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the `argument_default=` keyword argument to *`ArgumentParser`*. For example, to globally suppress attribute creation on *`parse_args()`* calls, we supply `argument_default=SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

Normally, when you pass an argument list to the *`parse_args()`* method of an *`ArgumentParser`*, it *recognizes abbreviations* of long options.

This feature can be disabled by setting `allow_abbrev` to `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
```

(繼續下一頁)

(繼續上一頁)

```
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

在 3.5 版被加入.

conflict_handler

ArgumentParser objects do not allow two actions with the same option string. By default, *ArgumentParser* objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Sometimes (e.g. when using *parents*) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value `'resolve'` can be supplied to the `conflict_handler=` argument of *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Note that *ArgumentParser* objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

add_help

By default, *ArgumentParser* objects add an option which simply displays the parser's help message. If `-h` or `--help` is supplied at the command line, the `ArgumentParser` help will be printed.

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help=` argument to *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

The help option is typically `-h/--help`. The exception to this is if the `prefix_chars=` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]
```

(繼續下一頁)

(繼續上一頁)

```
options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normally, when you pass an invalid argument list to the *parse_args()* method of an *ArgumentParser*, it will print a *message* to *sys.stderr* and exit with a status code of 2.

If the user would like to catch errors manually, the feature can be enabled by setting `exit_on_error` to `False`:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None,␣
↪default=None, type=<class 'int'>, choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

在 3.9 版被加入.

suggest_on_error

By default, when a user passes an invalid argument choice or subparser name, *ArgumentParser* will exit with error info and list the permissible argument choices (if specified) or subparser names as part of the error message.

If the user would like to enable suggestions for mistyped argument choices and subparser names, the feature can be enabled by setting `suggest_on_error` to `True`. Note that this only applies for arguments when the choices specified are strings:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.',
                                     suggest_on_error=True)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--action', 'sumn', 1, 2, 3])
tester.py: error: argument --action: invalid choice: 'sumn', maybe you meant 'sum'? (choose␣
↪from 'sum', 'max')
```

If you're writing code that needs to be compatible with older Python versions and want to opportunistically use `suggest_on_error` when it's available, you can set it as an attribute after initializing the parser instead of using the keyword argument:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
>>> parser.suggest_on_error = True
```

在 3.14 版被加入.

color

By default, the help message is printed in color using ANSI escape sequences. If you want plain text help messages, you can disable this in your local environment, or in the argument parser itself by setting `color` to `False`:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.',
...                                  color=False)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
```

(繼續下一頁)

(繼續上一頁)

```
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--help'])
```

在 3.14 版被加入.

17.1.2 add_argument() 方法

ArgumentParser.**add_argument**(*name or flags...*, *[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*][, *deprecated*])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- *name or flags* - Either a name or a list of option strings, e.g. `'foo'` or `'-f', '--foo'`.
- *action* - The basic type of action to be taken when this argument is encountered at the command line.
- *nargs* - The number of command-line arguments that should be consumed.
- *const* - A constant value required by some *action* and *nargs* selections.
- *default* - The value produced if the argument is absent from the command line and if it is absent from the namespace object.
- *type* - The type to which the command-line argument should be converted.
- *choices* - A sequence of the allowable values for the argument.
- *required* - Whether or not the command-line option may be omitted (optionals only).
- *help* - A brief description of what the argument does.
- *metavar* - A name for the argument in usage messages.
- *dest* - The name of the attribute to be added to the object returned by *parse_args()*.
- *deprecated* - Whether or not use of the argument is deprecated.

The following sections describe how each of these are used.

name or flags

The *add_argument()* method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to *add_argument()* must therefore be either a series of flags, or a simple argument name.

For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```
>>> parser.add_argument('bar')
```

When *parse_args()* is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
```

(繼續下一頁)

(繼續上一頁)

```
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

By default, `argparse` automatically handles the internal naming and display names of arguments, simplifying the process without requiring additional configuration. As such, you do not need to specify the *dest* and *metavar* parameters. The *dest* parameter defaults to the argument name with underscores _ replacing hyphens - . The *metavar* parameter defaults to the upper-cased name. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo-bar')
>>> parser.parse_args(['--foo-bar', 'FOO-BAR']
Namespace(foo_bar='FOO-BAR')
>>> parser.print_help()
usage:  [-h] [--foo-bar FOO-BAR]

optional arguments:
 -h, --help  show this help message and exit
 --foo-bar FOO-BAR
```

action

ArgumentParser objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by *parse_args()*. The `action` keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- `'store'` - This just stores the argument's value. This is the default action.
- `'store_const'` - This stores the value specified by the *const* keyword argument; note that the *const* keyword argument defaults to `None`. The `'store_const'` action is most commonly used with optional arguments that specify some sort of flag. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- `'store_true'` and `'store_false'` - These are special cases of `'store_const'` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - This appends each argument value to a list. It is useful for allowing an option to be specified multiple times. If the default value is a non-empty list, the parsed value will start with the default list's elements and any values from the command line will be appended after those default values. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append', default=['0'])
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['0', '1', '2'])
```

- `'append_const'` - This stores a list, and appends the value specified by the *const* keyword argument to the list; note that the *const* keyword argument defaults to `None`. The `'append_const'` action is typically useful when multiple arguments need to store constants to the same list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'extend'` - This stores a list and appends each item from the multi-value argument list to it. The `'extend'` action is typically used with the *nargs* keyword argument value `'+'` or `'*'`. Note that when *nargs* is `None` (the default) or `'?'`, each character of the argument string will be appended to the list. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

  在 3.8 版被加入.

- `'count'` - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  Note, the *default* will be `None` unless explicitly set to *0*.

- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See *`ArgumentParser`* for details of how the output is created.

- `'version'` - This expects a `version=` keyword argument in the *`add_argument()`* call, and prints version information and exits when invoked:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

你也可以傳遞一個*`Action`* 子類🄵（例如*`BooleanOptionalAction`*）或實作相同介面的其他物件。只有會消耗命令列引數的 action（例如 `'store'`、`'append'`、`'extend'` 或 `nargs` 不🄵零的自定義 action）可以被用於位置引數。

The recommended way to create a custom action is to extend *`Action`*, overriding the `__call__()` method and optionally the `__init__()` and `format_usage()` methods. You can also register custom actions using the *`register()`* method and reference them by their registered name.

An example of a custom action:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
```

(繼續下一頁)

(繼續上一頁)

```
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多詳情請見*Action*。

nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. See also 指定不明確的引數. The supported values are:

- `N` (an integer). `N` arguments from the command line will be gathered together into a list. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

- `'?'`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from *default* will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from *const* will be produced. Some examples to illustrate this:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  One of the more common uses of `nargs='?'` is to allow optional input and output files:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?')
  >>> parser.add_argument('outfile', nargs='?')
  >>> parser.parse_args(['input.txt', 'output.txt'])
  Namespace(infile='input.txt', outfile='output.txt')
  >>> parser.parse_args(['input.txt'])
  Namespace(infile='input.txt', outfile=None)
  >>> parser.parse_args([])
  Namespace(infile=None, outfile=None)
  ```

- `'*'`. All command-line arguments present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='*')
  >>> parser.add_argument('--bar', nargs='*')
  >>> parser.add_argument('baz', nargs='*')
  >>> parser.parse_args('a b --foo x y --bar 1 2'.split())
  Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
  ```

- `'+'`. Just like `'*'`, all command-line arguments present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the *action*. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced. Actions that do not consume command-line arguments (e.g. `'store_const'`) set `nargs=0`.

const

The `const` argument of *`add_argument()`* is used to hold constant values that are not read from the command line but are required for the various *`ArgumentParser`* actions. The two most common uses of it are:

- When *`add_argument()`* is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by *`parse_args()`*. See the *action* description for examples. If `const` is not provided to *`add_argument()`*, it will receive a default value of `None`.
- When *`add_argument()`* is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value from `const` will be used. See the *nargs* description for examples.

在 3.11 版的變更: `const=None` by default, including when `action='append_const'` or `action='store_const'`.

default

All optional arguments and some positional arguments may be omitted at the command line. The `default` keyword argument of *`add_argument()`*, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the target namespace already has an attribute set, the action *default* will not overwrite it:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any *type* conversion argument, if provided, before setting the attribute on the *`Namespace`* return value. Otherwise, the parser uses the value as is:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
```

(繼續下一頁)

(繼續上一頁)

```
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with *nargs* equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

For *required* arguments, the `default` value is ignored. For example, this applies to positional arguments with *nargs* values other than `?` or `*`, or optional arguments marked as `required=True`.

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

By default, the parser reads command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, such as a *float* or *int*. The `type` keyword for *add_argument()* allows any necessary type-checking and type conversions to be performed.

If the *type* keyword is used with the *default* keyword, the type converter is only applied if the default is a string.

The argument to `type` can be a callable that accepts a single string or the name of a registered type (see *register()*) If the function raises *ArgumentTypeError*, *TypeError*, or *ValueError*, the exception is caught and a nicely formatted error message is displayed. Other exception types are not handled.

Common built-in types and functions can be used as type converters:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('datapath', type=pathlib.Path)
```

User defined functions can be used as well:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

The *bool()* function is not recommended as a type converter. All it does is convert empty strings to `False` and non-empty strings to `True`. This is usually not what is desired.

In general, the `type` keyword is a convenience that should only be used for simple conversions that can only raise one of the three supported exceptions. Anything with more interesting error-handling or resource management should be done downstream after the arguments are parsed.

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the `type` keyword. A *`JSONDecodeError`* would not be well formatted and a *`FileNotFoundError`* exception would not be handled at all.

Even *`FileType`* has its limitations for use with the `type` keyword. If one argument uses *`FileType`* and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

For type checkers that simply check against a fixed set of values, consider using the *choices* keyword instead.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a sequence object as the *choices* keyword argument to *`add_argument()`*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Any sequence can be passed as the *choices* value, so *`list`* objects, *`tuple`* objects, and custom sequences are all supported.

Use of *`enum.Enum`* is not recommended because it is difficult to control its appearance in usage, help, and error messages.

Note that *choices* are checked after any *type* conversions have been performed, so objects in *choices* should match the *type* specified. This can make *choices* appear unfamiliar in usage, help, or error messages.

To keep *choices* user-friendly, consider a custom type wrapper that converts and formats values, or omit *type* and handle conversion in your application code.

Formatted choices override the default *metavar* which is normally derived from *dest*. This is usually what you want because the user never sees the *dest* parameter. If this display isn't desirable (perhaps because there are many choices), just specify an explicit *metavar*.

required

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to *`add_argument()`*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

As the example shows, if an option is marked as `required`, *`parse_args()`* will report an error if that option is not present at the command line.

> **備🄵**
>
> Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument.

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument *default*. The available specifiers include the program name, `%(prog)s` and most keyword arguments to *`add_argument()`*, e.g. `%(default)s`, `%(type)s`, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

options:
 -h, --help  show this help message and exit
```

As the help string supports %-formatting, if you want a literal `%` to appear in the help string, you must escape it as `%%`.

`argparse` supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

metavar

When *`ArgumentParser`* generates help messages, it needs some way to refer to each expected argument. By default, `ArgumentParser` objects use the *dest* value as the "name" of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

An alternative name can be specified with `metavar`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

Note that `metavar` only changes the *displayed* name - the name of the attribute on the *parse_args()* object is still determined by the *dest* value.

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

Most *ArgumentParser* actions add some value as an attribute of the object returned by *parse_args()*. The name of this attribute is determined by the `dest` keyword argument of *add_argument()*. For positional argument actions, `dest` is normally supplied as the first argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. *ArgumentParser* generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` allows a custom attribute name to be provided:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

During a project's lifetime, some arguments may need to be removed from the command line. Before removing them, you should inform your users that the arguments are deprecated and will be removed. The `deprecated` keyword argument of *`add_argument()`*, which defaults to `False`, specifies if the argument is deprecated and will be removed in the future. For arguments, if `deprecated` is `True`, then a warning will be printed to *`sys.stderr`* when the argument is used:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int, deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

在 3.13 版被加入.

Action 類🄵

`Action` classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to *`add_argument()`*.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

`Action` objects are used by an *`ArgumentParser`* to represent the information needed to parse a single argument from one or more strings from the command line. The `Action` class must accept the two positional arguments plus any keyword arguments passed to *`ArgumentParser.add_argument()`* except for the `action` itself.

Instances of `Action` (or return value of any callable to the `action` parameter) should have attributes `dest`, `option_strings`, `default`, `type`, `required`, `help`, etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__()`.

__call__(*parser*, *namespace*, *values*, *option_string=None*)

`Action` instances should be callable, so subclasses must override the `__call__()` method, which should accept four parameters:

- *parser* - The *`ArgumentParser`* object which contains this action.
- *namespace* - The *`Namespace`* object that will be returned by *`parse_args()`*. Most actions add an attribute to this object using *`setattr()`*.
- *values* - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the *type* keyword argument to *`add_argument()`*.
- *option_string* - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__()` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

format_usage()

`Action` subclasses can define a `format_usage()` method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

class argparse.**BooleanOptionalAction**

A subclass of *Action* for handling boolean flags with positive and negative options. Adding a single argument such as `--foo` automatically creates both `--foo` and `--no-foo` options, storing `True` and `False` respectively:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

在 3.9 版被加入.

17.1.3 parse_args() 方法

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to *add_argument()* determine exactly what objects are created and how they are assigned. See the documentation for `add_argument()` for details.

- *args* - List of strings to parse. The default is taken from *sys.argv*.
- *namespace* - An object to take the attributes. The default is a new empty *Namespace* object.

Option value syntax

The *parse_args()* method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

無效引數

While parsing the command line, *parse_args()* checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # 無效型🄵
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # 無效選項
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # 錯誤引數數量
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 - 的引數

The *parse_args()* method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The *parse_args()* method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with - and don't look like negative numbers, you can insert the pseudo-argument `'--'` which tells *parse_args()* that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

See also *the argparse howto on ambiguous arguments* for more details.

引數縮寫 (前綴匹配)

The *parse_args()* method *by default* allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting *allow_abbrev* to `False`.

`sys.argv` 之外

Sometimes it may be useful to have an *ArgumentParser* parse arguments other than those of *sys.argv*. This can be accomplished by passing a list of strings to *parse_args()*. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空間物件

class argparse.**Namespace**

Simple class used by default by *parse_args()* to create an object holding attributes and return it.

This class is deliberately simple, just an *object* subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, *vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an *ArgumentParser* assign attributes to an already existing object, rather than a new *Namespace* object. This can be achieved by specifying the `namespace=` keyword argument:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

17.1.4 Other utilities

子命令

ArgumentParser.**add_subparsers**(*[, *title*][, *description*][, *prog*][, *parser_class*][, *action*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of subcommands, for example, the `svn` program can invoke subcommands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such subcommands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any `ArgumentParser` constructor arguments, and returns an `ArgumentParser` object that can be modified as usual.

參數的解釋：

- *title* - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- *description* - description for the sub-parser group in help output, by default `None`
- *prog* - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- *parser_class* - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. *ArgumentParser*)
- *action* - the basic type of action to be taken when this argument is encountered at the command line
- *dest* - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- *required* - Whether or not a subcommand must be provided, by default `False` (added in 3.7)
- *help* - help for sub-parser group in help output, by default `None`
- *metavar* - string presenting available subcommands in help; by default it is `None` and presents subcommands in form {cmd1, cmd2, ..}

一些使用範例：

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='subcommand help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices=('X', 'Y', 'Z'), help='baz help')
```

(繼續下一頁)

(繼續上一頁)

```
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by *parse_args()* will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   subcommand help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The *add_subparsers()* method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands
```

(繼續下一頁)

(繼續上一頁)

```
  {foo,bar}   additional help
```

Furthermore, `add_parser()` supports an additional *aliases* argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

`add_parser()` supports also an additional *deprecated* argument, which allows to deprecate the subparser.

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

在 3.13 版被加入.

One particularly effective way of handling subcommands is to combine the use of the *`add_subparsers()`* method with calls to *`set_defaults()`* so that each subparser knows which Python function it should execute. For example:

```
>>> # subcommand functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let *parse_args()* do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the *add_subparsers()* call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

在 3.7 版的變更: New *required* keyword-only parameter.

在 3.14 版的變更: Subparser's *prog* is no longer affected by a custom usage message in the main parser.

FileType 物件

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

The *FileType* factory creates objects that can be passed to the type argument of *ArgumentParser.add_argument()*. Arguments that have *FileType* objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the *open()* function for more details):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.
↪FileIO name='raw.dat' mode='wb'>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into *sys.stdin* for readable *FileType* objects and *sys.stdout* for writable *FileType* objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

備🄵

If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. This can also clobber the output files. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

在 3.4 版的變更: Added the *encodings* and *errors* parameters.

在 3.14 版之後被🄵用.

Argument groups

ArgumentParser.**add_argument_group**(*title=None*, *description=None*, ***[, *argument_default*][, *conflict_handler*])

By default, *ArgumentParser* groups command-line arguments into "positional arguments" and "options" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The *add_argument_group()* method returns an argument group object which has an *add_argument()* method just like a regular *ArgumentParser*. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

The optional, keyword-only parameters *argument_default* and *conflict_handler* allow for finer-grained control of the behavior of the argument group. These parameters have the same meaning as in the *ArgumentParser* constructor, but apply specifically to the argument group rather than the entire parser.

Note that any arguments not in your user-defined groups will end up back in the usual ”positional arguments” and ”optional arguments” sections.

Deprecated since version 3.11, removed in version 3.14: Calling *add_argument_group()* on an argument group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

在 3.14 版之後被🄴用: Passing *prefix_chars* to *add_argument_group()* is now deprecated.

Mutual exclusion

`ArgumentParser.`**`add_mutually_exclusive_group`**(*required=False*)

Create a mutually exclusive group. `argparse` will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
```

(繼續下一頁)

(繼續上一頁)

```
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The *add_mutually_exclusive_group()* method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of *add_argument_group()*. However, a mutually exclusive group can be added to an argument group that has a title and description. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group description')
>>> exclusive_group = group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit

Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

Deprecated since version 3.11, removed in version 3.14: Calling *add_argument_group()* or *add_mutually_exclusive_group()* on a mutually exclusive group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

Parser defaults

ArgumentParser.**set_defaults**(***kwargs*)

Most of the time, the attributes of the object returned by *parse_args()* will be fully determined by inspecting the command-line arguments and the argument actions. *set_defaults()* allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the *add_subparsers()* method for an example of this type.

ArgumentParser.**get_default**(*dest*)

Get the default value for a namespace attribute, as set by either *add_argument()* or by *set_defaults()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

印出幫助訊息

In most typical applications, *parse_args()* will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.**print_usage**(*file=None*)

Print a brief description of how the *ArgumentParser* should be invoked on the command line. If *file* is None, *sys.stdout* is assumed.

ArgumentParser.**print_help**(*file=None*)

Print a help message, including the program usage and information about the arguments registered with the *ArgumentParser*. If *file* is None, *sys.stdout* is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.**format_usage**()

Return a string containing a brief description of how the *ArgumentParser* should be invoked on the command line.

ArgumentParser.**format_help**()

Return a string containing a help message, including the program usage and information about the arguments registered with the *ArgumentParser*.

Partial parsing

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

Sometimes a script only needs to handle a specific set of command-line arguments, leaving any unrecognized arguments for another script or program. In these cases, the *parse_known_args()* method can be useful.

This method works similarly to *parse_args()*, but it does not raise an error for extra, unrecognized arguments. Instead, it parses the known arguments and returns a two item tuple that contains the populated namespace and the list of any unrecognized arguments.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告

Prefix matching rules apply to *parse_known_args()*. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Customizing file parsing

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

Arguments that are read from a file (see the *fromfile_prefix_chars* keyword argument to the *ArgumentParser* constructor) are read one argument per line. *convert_arg_line_to_args()* can be overridden for fancier reading.

This method takes a single argument *arg_line* which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting methods

ArgumentParser.**exit**(*status=0*, *message=None*)

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* to *sys.stderr* before that. The user can override this method to handle these steps differently:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser.**error**(*message*)

This method prints a usage message, including the *message*, to *sys.stderr* and terminates the program with a status code of 2.

Intermixed parsing

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The *parse_intermixed_args()* and *parse_known_intermixed_args()* methods support this parsing style.

These parsers do not support all the `argparse` features, and will raise exceptions if unsupported features are used. In particular, subparsers, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between *parse_known_args()* and *parse_intermixed_args()*: the former returns `['2', '3']` as unparsed arguments, while the latter collects all the positionals into `rest`.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() returns a two item tuple containing the populated namespace and the list of remaining argument strings. *parse_intermixed_args()* raises an error if there are any remaining unparsed argument strings.

在 3.7 版被加入.

Registering custom types or actions

`ArgumentParser.`**`register`**(*registry_name*, *value*, *object*)

Sometimes it's desirable to use a custom string in error messages to provide more user-friendly output. In these cases, `register()` can be used to register custom actions or types with a parser and allow you to reference the type by their registered name instead of their callable name.

The `register()` method accepts three arguments - a *registry_name*, specifying the internal registry where the object will be stored (e.g., `action`, `type`), *value*, which is the key under which the object will be registered, and object, the callable to be registered.

The following example shows how to register a custom type with a parser:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.register('type', 'hexadecimal integer', lambda s: int(s, 16))
>>> parser.add_argument('--foo', type='hexadecimal integer')
_StoreAction(option_strings=['--foo'], dest='foo', nargs=None, const=None, default=None,
↪ type='hexadecimal integer', choices=None, required=False, help=None, metavar=None,␣
↪deprecated=False)
>>> parser.parse_args(['--foo', '0xFA'])
Namespace(foo=250)
>>> parser.parse_args(['--foo', '1.2'])
usage: PROG [-h] [--foo FOO]
PROG: error: argument --foo: invalid 'hexadecimal integer' value: '1.2'
```

17.1.5 例外

exception `argparse.`**`ArgumentError`**

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

exception `argparse.`**`ArgumentTypeError`**

Raised when something goes wrong converting a command line string to a type.

Guides and Tutorials

Argparse 教學

作者
Tshepang Mbambo

本教學旨在簡要介紹*`argparse`* 這個 Python 標准函式庫中推薦的命令列剖析模組。

> **備🄵**
>
> 標准函式庫包含另外兩個與命令列參數處理直接相關的函式庫：較低階的*`optparse`* 模組（可能需要更多程式碼來🄴給定應用程式設定，但也允許應用程式要求 `argparse` 不支援的行🄴），以及非常低階的*`getopt`*（專門用作 C 程式設計師可用的 `getopt()` 函式系列的等價）。雖然這個指南🄵未直接涵蓋這些模組，但 `argparse` 的許多核心概念最初來自於 `optparse`，因此本教學的某些部分也適用於 `optparse` 使用者。

概念

讓我們透過使用 **`ls`** 指令來展示我們將在本介紹教學中探索的功能類型：

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

我們可以從這四個命令中學到一些概念：

- **ls** 命令即便在□有任何選項的情□下執行仍非常有用。它預設顯示目前目□的□容。
- 如果我們想要看到比它預設提供更多的□容，我們也需要多告訴它一點。在本例中，我們希望它顯示不同的目□ `pypy`，我們做的是指定所謂的位置引數。之所以如此命名是因□程式應該只根據該值在命令列中出現的位置來知道如何處理該值。這個概念與 **cp** 這樣的指令更相關，其最基本的用法是 `cp SRC DEST`。第一個是你想要□□的位置，第二個是你想要□□過去的位置。
- 現在假設我們想要改變程式的行□。在我們的範例中，我們顯示每個檔案的更多資訊，而不僅是顯示檔案名稱。在這種情□下，`-l` 被稱□可選引數。
- 這是幫助文字的片段。它非常有用，因□當你遇到以前從未使用過的程式時，只需□讀其幫助文字即可了解它的工作原理。

基本用法

讓我們從一個非常簡單的例子開始，它（幾乎）什□都不做：

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

程式碼執行結果如下：

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

這是發生的事情：

- 執行不帶任何選項的□本不會在標準輸出中顯示任何□容。不太有用。
- 第二個開始能□顯現 *argparse* 模組的有用之處。我們幾乎什□也□做，但我們已經收到了一個很好的幫助訊息。
- `--help` 選項也可以縮寫□ `-h`，是我們能隨意獲得的唯一選項（即無需指定它）。指定任何其他□容都會導致錯誤。但即便如此，我們也還是輕松地獲得了有用的使用資訊。

位置引數的介紹

例如：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

執行這段程式碼：

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

這是會發生的事情：

- 我們新增了*add_argument()* 方法，我們用它來指定程式願意接受哪些命令列選項。在本例中，我將其命名🄵 `echo`，以便與其功能一致。
- 現在呼叫我們的程式時需要指定一個選項。
- *parse_args()* 方法實際上從指定的選項中回傳一些資料，在本例中🄵 `echo`。
- 該變數是某種形式的「魔法」，*argparse* 可以自由執行（即無需指定該值儲存在哪個變數中）。你還會注意到，它的名稱與提供給方法 `echo` 的字串引數相符。

但請注意，🄵管幫助顯示看起來不錯，但它目前還🄵有發揮出應有的用處。例如，我們看到 `echo` 作🄵位置引數，但除了猜測或🄵讀原始程式碼之外，我們不知道它的作用。那🄵，我們來讓它變得更有用一點：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

然後我們得到：

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

現在來做一些更有用處的事情：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
```

(繼續下一頁)

(繼續上一頁)

```
args = parser.parse_args()
print(args.square**2)
```

程式碼執行結果如下：

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

進展不太順利。這是因🄵，除非我們另有🄵明，*argparse* 會將我們給它的選項視🄵字串。因此，讓我們告訴*argparse* 將該輸入視🄵整數：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

程式碼執行結果如下：

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

順利進展。現在該程式甚至可以在繼續操作之前因錯誤的非法輸入而退出。

可選引數的介紹

到目前🄵止，我們一直在討論位置引數。我們來看看如何新增可選引數：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

接者是結果：

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

這是發生的事情：

- 程式被編寫🄵在指定 `--verbosity` 時顯示一些🄵容，🄵在未指定時不顯示任何🄵容。

- 🄵了表示該選項實際上是可選的，🄵使用它來執行程式🄵不會出現錯誤。請注意，預設情🄵下，如果未使用可選引數，則相關變數（在本例中🄵 `args.verbosity`）將被賦予 `None` 作🄵值，這就是它未能通過 `if` 陳述式真值測試的原因。
- 幫助訊息有點不同。
- 當使用 `--verbosity` 選項時必須要指定一些值，任何值都可以。

在上面的例子中，`--verbosity` 接受任意的整數，但對我們的程式來🄵只接受兩個輸入值，`True` 或 `False`。所以我們來修改一下程式碼使其符合：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

接者是結果：

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

這是發生的事情：

- 這個選項現在更像是一個旗標，而不是需要值的東西。我們甚至更改了選項的名稱以符合這個想法。請注意，我們現在指定一個新的關鍵字 `action`，🄵🄵其指定值 `"store_true"`。這意味著，如果指定了該選項，則將值 `True` 指派給 `args.verbose`。不指定它代表🄵 `False`。
- 當你指定一個值時，它會本著旗標的實際精神來抱怨。
- 請注意不同的幫助文字。

短選項

如果你熟悉命令列用法，你會注意到我尚未提及選項的簡短版本。這很簡單：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

而這🄵：

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

options:
```

(繼續下一頁)

(繼續上一頁)

```
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

請注意，新功能也反映在幫助文字中。

組合位置引數和可選引數

我們的程式的🄴雜性不斷增加：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

然後現在的輸出結果：

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- 我們帶回了位置引數，因而被抱怨。
- 請注意，順序🄴不重要。

我們讓這個程式擁有多個訊息詳細級🄴 (verbosity) 之值的能力，🄴實際使用它們：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

接者是結果：

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
```

(繼續下一頁)

(繼續上一頁)

```
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

除了最後一個外都看起來正常，它透露了我們程式中的一個錯誤。我們可透過限制 `--verbosity` 選項可以接受的值來修復它：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

接者是結果：

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

請注意，更改也會反映在錯誤訊息和幫助字串中。

現在，讓我們使用另一種常見方法來玩玩訊息詳細級🄴。它也與 CPython 執行檔處理其自身訊息詳細級🄴引數的方式相符（請見 `python --help` 的輸出）：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我們已經介紹過另一個操作"count" 用來計算指定的選項出現的次數。

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- 是的，現在它更像是我們上一版🄵本中的旗標（類似於 `action="store_true"`），這應該可以解釋抱怨的原因。
- 它的行🄵也類似"store_true" 操作。
- 現在這🄵示範了"count" 動作的作用。你可能以前見過這種用法。
- 如果你不指定 `-v` 旗標，則該旗標被視🄵具有 `None` 值。
- 正如預期的那樣，指定長形式旗標，我們應該得到相同的輸出。
- 遺憾的是，我們的幫助輸出對於我們🄵本獲得的新功能🄵🄵有提供太多資訊，但我們都可以透過改進🄵本的文件來解🄵這個問題（例如：透過 `help` 關鍵字引數）。
- 最後的輸出透露了我們程式中的一個錯誤。

讓我們來解🄵問題：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

這就是它給出的：

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
```

(繼續下一頁)

(繼續上一頁)

```
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

- 第一次輸出順利進行，🄵修復了我們之前遇到的錯誤。也就是🄵，我們希望任何 >= 2 的值都盡可能詳細。
- 第三個輸出不太好。

我們來修復這個錯誤：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我們剛剛引入了另一個關鍵字 `default`。我們將其設🄵 `0`，以便使其與其他 int 值進行比較。請記住，預設情🄵下，如果未指定可選引數，它將獲得 `None` 值，🄵且不能與 int 值進行比較（因此會出現*`TypeError`*例外）。

而且：

```
$ python prog.py 4
16
```

僅憑我們迄今🄵止所學到的知識就可以做到很多事情了，不過其實這樣只有學到一點皮毛而已。*`argparse`* 模組非常🄵大，在結束本教學之前我們會對它進行更多探索。

更進階一點

如果我們想擴充我們的小程式來執行其他次方的運算，而不僅是平方：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

結果：

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

請注意，到目前🄴止，我們一直在使用詳細級🄴來更改顯示的文字。以下範例使用詳細級🄴來顯示更多文字：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

結果：

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

指定不明確的引數

當🄴定一個引數是位置引數還是引數會有歧義，可以使用 `--` 來告訴*parse_args()* 之後的所有🄴容都是位置引數：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
```

(繼續下一頁)

(繼續上一頁)

```
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

相互沖突的選項

到目前🄴止，我們一直在使用*argparse.ArgumentParser* 實例的兩種方法。讓我們介紹第三個，*add_mutually_exclusive_group()*，它允許我們指定彼此衝突的選項。我們還可以更改程式的其余部分，以使得新功能更有意義：我們將引入 `--quiet` 選項，該選項與 `--verbose` 選項相反：

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

我們的程式現在更簡單了，我們因🄴功能展示失去了一些功能，但無論如何，以下這是輸出：

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

這應該很容易理解。我新增了最後一個輸出，以便看到所獲得的靈活性，即可以混合長形式與短形式選項。

在我們結束之前，你可能想告訴使用者你的程式的主要目的，以防他們不知道：

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y
```

(繼續下一頁)

(繼續上一頁)

```
if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

請注意用法文字中的細微差🄵。注意 `[-v | -q]`，它告訴我們可以使用 `-v` 或 `-q`，但不能同時使用：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

如何翻譯 argparse 輸出

argparse 模組的輸出，例如幫助文字和錯誤訊息，都可以透過使用*gettext* 模組進行翻譯。這允許應用程式能輕松本地化*argparse* 生成的訊息。另請參🄵*Internationalizing your programs and modules*。

例如，在此*argparse* 輸出中：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

字串 `usage:`、`positional arguments:`、`options:` 和 `show this help message and exit` 都是可被翻譯的。

🄵了翻譯這些字串，必須先將它們提取到 `.po` 檔案中。例如，使用 Babel 🄵執行下列命令：

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

此命令將從*argparse* 模組中提取出所有可翻譯的字串，🄵將它們輸出到名🄵 `messages.po` 的檔案中。這個指令假設你的 Python 是安裝在 `/usr/lib` 中。

你可以使用以下🄵本找到*argparse* 模組在系統上的位置：

```
import argparse
print(argparse.__file__)
```

一旦翻譯了 `.po` 檔案中的訊息🄵使用*gettext* 安裝了翻譯，*argparse* 將能🄵顯示翻譯後的訊息。

若要在*argparse* 輸出中翻譯你自己的字串，請使用*gettext*。

自定義型🄵轉🄵器

The *argparse* module allows you to specify custom type converters for your command-line arguments. This allows you to modify user input before it's stored in the *argparse.Namespace*. This can be useful when you need to pre-process the input before it is used in your program.

When using a custom type converter, you can use any callable that takes a single string argument (the argument value) and returns the converted value. However, if you need to handle more complex scenarios, you can use a custom action class with the **action** parameter instead.

For example, let's say you want to handle arguments with different prefixes and process them accordingly:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print(args)
```

結果：

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

在這個範例當中，我們：

- Created a parser with custom prefix characters using the `prefix_chars` parameter.
- Defined two arguments, `-a` and `+a`, which used the `type` parameter to create custom type converters to store the value in a tuple with the prefix.

Without the custom type converters, the arguments would have treated the `-a` and `+a` as the same argument, which would have been undesirable. By using custom type converters, we were able to differentiate between the two arguments.

結論

argparse 模組提供的功能比此篇🄵容的要多得多。它的文件非常詳細與透徹🄵有很多範例。讀完本教學後，你應該可以輕松消化它們，而不會感到不知所措。

將 `optparse` 程式碼遷移到 `argparse`

argparse 模組提供了一些高階功能，這些功能在*optparse* 模組中🄵未原生提供，包括：

- 處理位置引數。
- 支援子命令。
- 允許替代選項前綴，如 `+` 和 `/`。
- 處理零或多個 (zero-or-more) 和一個或多個 (and one-or-more) 樣式的引數。
- 🄵生更多資訊的使用訊息。
- 🄵自訂 `type` 和 `action` 提供了一個更簡單的介面。

最初*argparse* 模組試圖保持與*optparse* 的相容性，但在基礎設計上的存在差🄵 -- 支援宣告式 (declarative) 命令列選項處理 (同時將位置引數處理留給應用程式的程式碼) 和在宣告式介面中支援命名選項 (named options) 和位置引數 -- 代表 API 隨著時間的推移已經與 `optparse` API 分歧。

如*選擇一個命令列參數剖析函式庫* 中所述，目前使用*optparse* 🄵對其運作方式滿意的應用程式可以繼續使用 `optparse`。

在🄵定是否遷移之前，應用程式開發人員應該先檢🄵該段落中描述的🄵在行🄵差🄵清單，來🄵定是否值得遷移。

對於選擇從*optparse* 遷移到*argparse* 的應用程式，以下建議應會有所幫助：

- 將所有*optparse.OptionParser.add_option()* 呼叫替🄵🄵*ArgumentParser.add_argument()* 呼叫。
- 將 `(options, args) = parser.parse_args()` 替🄵🄵 `args = parser.parse_args()`，🄵🄵位置引數新增額外的*ArgumentParser.add_argument()* 呼叫。請記住，以前稱🄵 `options` 的東西，在*argparse* 情境中現在稱🄵 `args`。
- 使用*parse_intermixed_args()* 來替🄵掉*optparse.OptionParser.disable_interspersed_args()*，而不是使用*parse_args()*。
- 將回呼動作和 `callback_*` 關鍵字引數替🄵🄵 `type` 或 `action` 引數。
- 將 `type` 關鍵字引數的字串名稱替🄵🄵相應的類型物件 (例如 int、float、complex 等)。
- 將*optparse.Values* 替🄵🄵*Namespace*，🄵將*optparse.OptionError* 和*optparse.OptionValueError* 替🄵🄵*ArgumentError*。
- 將隱式引數的字串，如 `%default` 或 `%prog` 替🄵🄵使用字典來格式化字串的標準 Python 語法，即 `%(default)s` 和 `%(prog)s`。
- 將 OptionParser 建構函式的 `version` 引數替🄵🄵呼叫 `parser.add_argument('--version', action='version', version='<the version>')`。

17.2 `optparse` --- 命令列選項剖析器

原始碼： Lib/optparse.py

17.2.1 選擇一個命令列參數剖析函式庫

標準函式庫包含三個命令列引數剖析函式庫：

- *getopt*: a module that closely mirrors the procedural C `getopt` API. Included in the standard library since before the initial Python 1.0 release.
- *optparse*: a declarative replacement for `getopt` that provides equivalent functionality without requiring each application to implement its own procedural option parsing logic. Included in the standard library since the Python 2.3 release.
- *argparse*: a more opinionated alternative to `optparse` that provides more functionality by default, at the expense of reduced application flexibility in controlling exactly how arguments are processed. Included in the standard library since the Python 2.7 and Python 3.2 releases.

In the absence of more specific argument parsing design constraints, *argparse* is the recommended choice for implementing command line applications, as it offers the highest level of baseline functionality with the least application level code.

getopt is retained almost entirely for backwards compatibility reasons. However, it also serves a niche use case as a tool for prototyping and testing command line argument handling in `getopt`-based C applications.

optparse should be considered as an alternative to *argparse* in the following cases:

- an application is already using *optparse* and doesn't want to risk the subtle behavioural changes that may arise when migrating to *argparse*
- the application requires additional control over the way options and positional parameters are interleaved on the command line (including the ability to disable the interleaving feature completely)
- the application requires additional control over the incremental parsing of command line elements (while `argparse` does support this, the exact way it works in practice is undesirable for some use cases)

- the application requires additional control over the handling of options which accept parameter values that may start with - (such as delegated options to be passed to invoked subprocesses)
- the application requires some other command line parameter processing behavior which `argparse` does not support, but which can be implemented in terms of the lower level interface offered by `optparse`

These considerations also mean that *optparse* is likely to provide a better foundation for library authors writing third party command line argument processing libraries.

As a concrete example, consider the following two command line argument parsing configurations, the first using `optparse`, and the second using `argparse`:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

The most obvious difference is that in the `optparse` version, the non-option arguments are processed separately by the application after the option processing is complete. In the `argparse` version, positional arguments are declared and processed in the same way as the named options.

However, the `argparse` version will also handle some parameter combination differently from the way the `optparse` version would handle them. For example (amongst other differences):

- supplying `-o -v` gives `output="-v"` and `verbose=False` when using `optparse`, but a usage error with `argparse` (complaining that no value has been supplied for `-o/--output`, since `-v` is interpreted as meaning the verbosity flag)
- similarly, supplying `-o --` gives `output="--"` and `args=()` when using `optparse`, but a usage error with `argparse` (also complaining that no value has been supplied for `-o/--output`, since `--` is interpreted as terminating the option processing and treating all remaining values as positional arguments)
- supplying `-o=foo` gives `output="=foo"` when using `optparse`, but gives `output="foo"` with `argparse` (since `=` is special cased as an alternative separator for option parameter values)

Whether these differing behaviors in the `argparse` version are considered desirable or a problem will depend on the specific command line application use case.

也參考

click is a third party argument processing library (originally based on `optparse`), which allows command line applications to be developed as a set of decorated command implementation functions.

Other third party libraries, such as typer or msgspec-click, allow command line interfaces to be specified in ways that more effectively integrate with static checking of Python type annotations.

17.2.2 Introduction

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the minimalist *getopt* module. *optparse* uses a more declarative style of command-line parsing: you create an instance of *OptionParser*, populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by *parse_args()* based on user-supplied command-line values. When *parse_args()* returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

17.2.3 背景

optparse was explicitly designed to encourage the creation of programs with straightforward command-line interfaces that follow the conventions established by the `getopt()` family of functions available to C developers. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, reading this section will allow you to acquaint yourself with them.

術語

引數
a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

選項
an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

選項引數
an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

位置引數
something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option
an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`---all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user---most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply---use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options---the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

17.2.4 教學

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to *parse_args()*, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

parse_args() 回傳兩個值:

- `options`, an object containing values for all of your options---e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable---for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

舉例來🄵:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to *parse_args()*, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

會印出 `42`。

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options---set a variable to true or false when a particular option is seen---are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values---see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are:

`"store_const"`
: store a constant value, pre-set via *Option.const*

`"append"`
: append this option's argument to a list

`"count"`
: increment a counter by one

`"callback"`
: call a specified function

These are covered in section *Reference Guide*, and section *Option Callbacks*.

預設值

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling *`parse_args()`*:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

`optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *`help`* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *`optparse`* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *`optparse`* exits after printing the help text.)

There's a lot going on here to help *`optparse`* generate the best possible help message:

- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping---*optparse* takes care of wrapping lines and making the help output look good.
- options that take a value indicate this fact in their automatically generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want---for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string---*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**`(`*`opt_str`*`)`

Return the *`OptionGroup`* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *`OptionGroup`*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**`(`*file=None*`)`

Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**`()`

Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either *optparse.OptionError* or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

17.2.5 Reference Guide

Creating the parser

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage（預設值：`"%prog [options]"`）
The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list（預設值：`[]`）
A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (default: optparse.Option)
Class to use when adding options to the parser in *add_option()*.

version（預設值：`None`）
A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler（預設值：`"error"`）
Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description（預設值：**None**）
A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: a new **IndentedHelpFormatter**)
An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option（預設值：**True**）
If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog
The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog（預設值：**None**）
A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section 教學. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

定義選項

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an *Option* instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**add_option**(*option*)

`OptionParser.`**add_option**(**opt_str*, *attr=value*, ...)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an *OptionError* exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are:

"store"
store this option's argument (default)

"store_const"
store a constant value, pre-set via *Option.const*

"store_true"
store `True`

"store_false"
store `False`

"append"
append this option's argument to a list

"append_const"
append a constant value to a list, pre-set via *Option.const*

"count"
increment a counter by one

"callback"
call a specified function

"help"
print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options`, which is an instance of *optparse.Values*.

class optparse.**Values**

An object holding parsed argument names and values as attributes. Normally created by calling when calling *OptionParser.parse_args()*, and can be overridden by a custom subclass passed to the *values* argument of *OptionParser.parse_args()* (as described in 剖析引數).

Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

例如🄵，當你呼叫：

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

選項屬性

class optparse.**Option**

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with *OptionParser.add_option()* rather than directly, and can be overridden by a custom class via the *option_class* argument to *OptionParser*.

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises *OptionError*.

Option.**action**

(預設值: `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

Option.**type**

(預設值: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented *here*.

Option.**dest**

(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

Option.**default**

The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

Option.**nargs**

(預設值: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**

For actions that store a constant value, the constant value to store.

Option.**choices**

For options of type `"choice"`, the list of strings the user may choose from.

Option.**callback**

For options with action `"callback"`, the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**

Option.**callback_kwargs**

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

Option.**help**

Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.**metavar**

(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section 教學 for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

  If *type* is not supplied, it defaults to `"string"`.

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

  範例:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  As it parses the command line

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  optparse will set

  ```
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- `"store_const"` [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  範例:

  ```
  parser.add_option("-q", "--quiet",
                    action="store_const", const=0, dest="verbose")
  parser.add_option("-v", "--verbose",
                    action="store_const", const=1, dest="verbose")
  parser.add_option("--noisy",
                    action="store_const", const=2, dest="verbose")
  ```

  If `--noisy` is seen, *optparse* will set

  ```
  options.verbose = 2
  ```

- `"store_true"` [relevant: *dest*]

  A special case of `"store_const"` that stores `True` to *dest*.

- `"store_false"` [relevant: *dest*]

  Like `"store_true"`, but stores `False`.

  範例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  範例:

  ```
  parser.add_option("-t", "--tracks", action="append", type="int")
  ```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

  ```
  options.tracks = []
  options.tracks.append(int("3"))
  ```

  If, a little later on, --tracks=4 is seen, it does:

  ```
  options.tracks.append(int("4"))
  ```

  The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

  ```
  >>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
  >>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
  >>> opts.files
  ['~/.mypkg/defaults', 'overrides.mypkg']
  ```

- "append_const" [required: *const*; relevant: *dest*]

  Like "store_const", but the value *const* is appended to *dest*; as with "append", *dest* defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  範例:

  ```
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time -v is seen on the command line, *optparse* does the equivalent of:

  ```
  options.verbosity = 0
  options.verbosity += 1
  ```

  Every subsequent occurrence of -v results in

  ```
  options.verbosity += 1
  ```

- "callback" [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

更多細節請見*Option Callbacks*。

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *`help`* string passed to every option.

  If no *`help`* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

  `optparse` automatically adds a *`help`* option to all OptionParsers, so you do not normally need to create one.

  範例：

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  If *`optparse`* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

  After printing the help message, *`optparse`* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *`help`* options, you will rarely create `version` options, since *`optparse`* automatically adds them when needed.

Standard option types

`optparse` has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *`int()`* with the appropriate base (2, 8, 10, or 16). If this fails, so will *`optparse`*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises *OptionValueError* if an invalid string is given.

剖析引數

The whole point of creating and populating an OptionParser is to call its *parse_args()* method.

`OptionParser.`**`parse_args`**(*args=None*, *values=None*)

Parse the command-line options found in *args*.

The input parameters are

args
: the list of arguments to process (default: `sys.argv[1:]`)

values
: a *Values* object to store option arguments in (default: a new instance of *Values*) -- if you give an existing object, the option defaults will not be initialized on it

and the return value is a pair `(options, args)` where

options
: the same object that was passed in as *values*, or the `optparse.Values` instance created by *optparse*

args
: the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by *parse_args()*.

If *parse_args()* encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**()

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**()

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser.**get_option**(*opt_str*)

Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

OptionParser.**has_option**(*opt_str*)

Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

OptionParser.**remove_option**(*opt_str*)

If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (default)
assume option conflicts are a programming error and raise *OptionConflictError*

"resolve"
resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

其他方法

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**(*usage*)

Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**(*file=None*)

Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**()

Same as *`print_usage()`* but returns the usage string instead of printing it.

`OptionParser.`**`set_defaults`**(*dest=value*, ...)

Set default values for several option destinations at once. Using *`set_defaults()`* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *`set_defaults()`*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

17.2.6 Option Callbacks

When *`optparse`*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *`optparse`* or define a callback option. Extending *`optparse`* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments---the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type
: has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs
: also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args
: a tuple of extra positional arguments to pass to the callback

callback_kwargs
: a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option
: is the Option instance that's calling the callback

opt_str
: is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string---e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value
: is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser
: is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

  parser.largs
  : the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor

option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of *parse_args()*.)

parser.rargs
the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values
the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args
is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs
is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise *OptionValueError* if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to reuse this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
```

(繼續下一頁)

(繼續上一頁)

```
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there---you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard `"store"` action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []
```

(繼續下一頁)

(繼續上一頁)

```
    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

17.2.7 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s *Option* class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**

A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an *Option* instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by *OptionParser.parse_args()*, or be passed to a callback as the `value` parameter.

Your type-checking function should raise *OptionValueError* if it encounters any problems. *OptionValueError* takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions
: actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions
: actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**`ACTIONS`**
: All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
: "store" actions are additionally listed here.

Option.**TYPED_ACTIONS**

”typed” actions are additionally listed here.

Option.**ALWAYS_TYPED_ACTIONS**

Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns `value`. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

17.2.8 例外

exception optparse.**OptionError**

Raised if an *Option* instance is created with invalid or inconsistent arguments.

exception optparse.**OptionConflictError**

Raised if conflicting options are added to an *OptionParser*.

exception optparse.**OptionValueError**

Raised if an invalid option value is encountered on the command line.

exception optparse.**BadOptionError**

Raised if an invalid option is passed on the command line.

exception optparse.**AmbiguousOptionError**

Raised if an ambiguous option is passed on the command line.

17.3 getpass --- 可🄵式密碼輸入工具

原始碼： Lib/getpass.py

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

getpass 模組 (module) 提供了兩個函式：

getpass.**getpass**(*prompt='Password: '*, *stream=None*, ***, *echo_char=None*)

提示使用者輸入一個密碼且不會有回音 (echo)。使用者會看到字串 *prompt* 作🄵提示，其預設值🄵 `'Password: '`。在 Unix 上，如有必要的話會使用替🄵錯誤處理函式 (replace error handler) 寫入到類檔案物件 (file-like object) *stream* 中。*stream* 預設🄵主控終端機 (controlling terminal) (`/dev/tty`)，如果不可用則🄵 `sys.stderr` (此引數在 Windows 上會被忽略)。

The *echo_char* argument controls how user input is displayed while typing. If *echo_char* is `None` (default), input remains hidden. Otherwise, *echo_char* must be a single printable ASCII character and each typed character is replaced by it. For example, `echo_char='*'` will display asterisks instead of the actual input.

如果無回音輸入 (echo-free input) 無法使用則 getpass() 將回退🄵印出一條警告訊息到 *stream*，🄵從 `sys.stdin` 讀取且同時發出*GetPassWarning*。

> **備🄵**
>
> 如果你從 IDLE 🄵部呼叫 getpass，輸入可能會在你🄵動 IDLE 的終端機中完成，而非在 IDLE 視窗中。

> **備🄵**
>
> On Unix systems, when *echo_char* is set, the terminal will be configured to operate in *noncanonical mode*. In particular, this means that line editing shortcuts such as `Ctrl+U` will not work and may insert unexpected characters into the input.

在 3.14 版的變更: Added the *echo_char* parameter for keyboard feedback.

exception getpass.**GetPassWarning**

當密碼輸入可能被回音時會發出的*UserWarning* 子類🄵。

getpass.**getuser**()

回傳使用者的" 登入名稱"。

此函式會按順序檢查環境變數 LOGNAME、USER、LNAME 和 USERNAME，🄵回傳其中第一個被設定成非空字串的值。如果均未設定，則在支援*pwd* 模組的系統上將會回傳來自密碼資料庫的登入名稱，否則將引發一個*OSError* 例外。

大部分情🄵下，此函式應該要比*os.getlogin()* 優先使用。

在 3.13 版的變更: 在過去，除了*OSError* 外還會引發各種例外。

17.4 fileinput --- 逐列🄵代多個輸入串流

原始碼： Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see *open()*.

The typical use is:

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If a filename is '-', it is also replaced by sys.stdin and the optional arguments *mode* and *openhook* are ignored. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *input()* or *FileInput*. If an I/O error occurs during opening or reading a file, *OSError* is raised.

在 3.3 版的變更: *IOError* 曾經被引發；現在它是一個*OSError* 的🄵名。

If sys.stdin is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using sys.stdin.seek(0)).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to *fileinput.input()* or *FileInput()*. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. If *encoding* and/or *errors* are specified, they will be passed to the hook as additional keyword arguments. This module provides a *hook_compressed()* to support compressed files.

The following function is the primary interface of this module:

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Create an instance of the *FileInput* class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the *FileInput* class.

The *FileInput* instance can be used as a context manager in the with statement. In this example, *input* is closed after the with statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt'), encoding="utf-8") as f:
    for line in f:
        process(line)
```

在 3.2 版的變更: Can be used as a context manager.

在 3.8 版的變更: The keyword parameters *mode* and *openhook* are now keyword-only.

在 3.10 版的變更: The keyword-only parameter *encoding* and *errors* are added.

The following functions use the global state created by `fileinput.input()`; if there is no active state, `RuntimeError` is raised.

`fileinput.`**`filename`**`()`

Return the name of the file currently being read. Before the first line has been read, returns `None`.

`fileinput.`**`fileno`**`()`

Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns `-1`.

`fileinput.`**`lineno`**`()`

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

`fileinput.`**`filelineno`**`()`

Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

`fileinput.`**`isfirstline`**`()`

Return `True` if the line just read is the first line of its file, otherwise return `False`.

`fileinput.`**`isstdin`**`()`

Return `True` if the last line was read from `sys.stdin`, otherwise return `False`.

`fileinput.`**`nextfile`**`()`

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

`fileinput.`**`close`**`()`

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

`class` `fileinput.`**`FileInput`**`(`*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*`)`

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it is *iterable* and has a `readline()` method which returns the next input line. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With *mode* you can specify which file mode will be passed to `open()`. It must be one of `'r'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

You can specify *encoding* and *errors* that is passed to `open()` or *openhook*.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

在 3.2 版的變更: Can be used as a context manager.

在 3.8 版的變更: The keyword parameter *mode* and *openhook* are now keyword-only.

在 3.10 版的變更: The keyword-only parameter *encoding* and *errors* are added.

在 3.11 版的變更: The `'rU'` and `'U'` modes and the `__getitem__()` method have been removed.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to *`fileinput.input()`* or to the *`FileInput`* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

`fileinput.`**`hook_compressed`**(*filename*, *mode*, *, *encoding=None*, *errors=None*)

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the *`gzip`* and *`bz2`* modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using *`open()`* without any decompression).

The *encoding* and *errors* values are passed to *`io.TextIOWrapper`* for compressed files and open for normal files.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

在 3.10 版的變更: The keyword-only parameter *encoding* and *errors* are added.

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)

Returns a hook which opens each file with *`open()`*, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

在 3.6 版的變更: 新增可選參數 *errors*。

在 3.10 版之後被🄵用: This function is deprecated since *`fileinput.input()`* and *`FileInput`* now have *encoding* and *errors* parameters.

17.5 `curses` --- 字元儲存格顯示的終端處理

原始碼： Lib/curses

The *`curses`* module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

備🄵

Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

也參考

curses.ascii 模組
: Utilities for working with ASCII characters, regardless of your locale settings.

curses.panel 模組
: A panel stack extension that adds depth to curses windows.

curses.textpad 模組
: Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto
: Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

17.5.1 函式

curses 模組定義了以下例外：

exception curses.**error**

Exception raised when a curses library function returns an error.

> **備🄵**
>
> Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to *A_NORMAL*.

curses 模組定義了以下函式：

curses.**assume_default_colors**(*fg*, *bg*, /)

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application.

- Assign terminal default foreground/background colors to color number `-1`. So `init_pair(x, COLOR_RED, -1)` will initialize pair *x* as red on default background and `init_pair(x, -1, COLOR_BLUE)` will initialize pair *x* as default foreground on blue.
- Change the definition of the color-pair `0` to `(fg, bg)`.

在 3.14 版被加入.

curses.**baudrate**()

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.**beep**()

Emit a short attention sound.

curses.**can_change_color**()

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

curses.**cbreak**()

Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first *raw()* then *cbreak()* leaves the terminal in cbreak mode.

curses.**color_content**(*color_number*)

Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS - 1`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.**color_pair**(*pair_number*)

Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. This attribute value can be combined with *A_STANDOUT*, *A_REVERSE*, and the other A_* attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)

Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()

Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()

Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)

Insert an *ms* millisecond pause in output.

curses.**doupdate**()

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The *doupdate()* ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a *noutrefresh()* call after write operations such as *addstr()* have been performed on a window. The normal *refresh()* call is simply noutrefresh() followed by doupdate(); if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single doupdate().

curses.**echo**()

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()

De-initialize the library, and return terminal to normal status.

curses.**erasechar**()

Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()

The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()

After *getch()* returns *KEY_MOUSE* to signal a mouse event, this method should be called to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of

one or more of the following constants, where *n* is the button number from 1 to 5: *BUTTONn_PRESSED*, *BUTTONn_RELEASED*, *BUTTONn_CLICKED*, *BUTTONn_DOUBLE_CLICKED*, *BUTTONn_TRIPLE_CLICKED*, *BUTTON_SHIFT*, *BUTTON_CTRL*, *BUTTON_ALT*.

在 3.10 版的變更: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

`curses.`**`getsyx`**`()`

Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If *leaveok* is currently `True`, then return `(-1, -1)`.

`curses.`**`getwin`**`(`*file*`)`

Read window related data stored in the file by an earlier *window.putwin()* call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.`**`has_colors`**`()`

Return `True` if the terminal can display colors; otherwise, return `False`.

`curses.`**`has_extended_color_support`**`()`

Return `True` if the module supports extended colors; otherwise, return `False`. Extended color support allows more than 256 color pairs for terminals that support more than 16 colors (e.g. xterm-256color).

Extended color support requires ncurses version 6.1 or later.

在 3.10 版被加入.

`curses.`**`has_ic`**`()`

Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_il`**`()`

Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_key`**`(`*ch*`)`

Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

`curses.`**`halfdelay`**`(`*tenths*`)`

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use *nocbreak()* to leave half-delay mode.

`curses.`**`init_color`**`(`*color_number*, *r*, *g*, *b*`)`

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS - 1`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When *init_color()* is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if *can_change_color()* returns `True`.

`curses.`**`init_pair`**`(`*pair_number*, *fg*, *bg*`)`

Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair can only be changed by *use_default_colors()* and *assume_default_colors()*). The value of *fg* and *bg* arguments must be between `0` and `COLORS - 1`, or, after calling `use_default_colors()` or `assume_default_colors()`, `-1`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.`**`initscr`**`()`

Initialize the library. Return a *window* object which represents the whole screen.

> **備𝔽**
>
> If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

curses.**is_term_resized**(*nlines*, *ncols*)

Return `True` if *resize_term()* would modify the window structure, `False` otherwise.

curses.**isendwin**()

Return `True` if *endwin()* has been called (that is, the curses library has been deinitialized).

curses.**keyname**(*k*)

Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128--255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

curses.**killchar**()

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**longname**()

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()*.

curses.**meta**(*flag*)

If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

curses.**mouseinterval**(*interval*)

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 milliseconds, or one fifth of a second.

curses.**mousemask**(*mousemask*)

Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

curses.**napms**(*ms*)

Sleep for *ms* milliseconds.

curses.**newpad**(*nlines*, *ncols*)

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The *refresh()* and *noutrefresh()* methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.**newwin**(*nlines*, *ncols*)

curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a new *window*, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.**nl**()

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.**nocbreak**()

Leave cbreak mode. Return to normal "cooked" mode with line buffering.

curses.**noecho**()

Leave echo mode. Echoing of input characters is turned off.

curses.**nonl**()

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.**noqiflush**()

When the `noqiflush()` routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.**noraw**()

Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.**pair_content**(*pair_number*)

Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `0` and `COLOR_PAIRS - 1`.

curses.**pair_number**(*attr*)

Return the number of the color-pair set by the attribute value *attr*. *color_pair()* is the counterpart to this function.

curses.**putp**(*str*)

Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of *putp()* always goes to standard output.

curses.**qiflush**([*flag*])

If *flag* is `False`, the effect is the same as calling *noqiflush()*. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

curses.**raw**()

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.**reset_prog_mode**()

Restore the terminal to "program" mode, as previously saved by *def_prog_mode()*.

curses.**reset_shell_mode**()

Restore the terminal to "shell" mode, as previously saved by *def_shell_mode()*.

curses.**resetty**()

Restore the state of the terminal modes to what it was at the last call to *savetty()*.

curses.**resize_term**(*nlines*, *ncols*)

Backend function used by *resizeterm()*, performing most of the work; when resizing the windows, *resize_term()* blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The `resize_term()` function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.**resizeterm**(*nlines*, *ncols*)

Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()

Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**get_escdelay**()

Retrieves the value set by *set_escdelay()*.

在 3.9 版被加入.

curses.**set_escdelay**(*ms*)

Sets the number of milliseconds to wait after reading an escape character, to distinguish between an individual escape character entered on the keyboard from escape sequences sent by cursor and function keys.

在 3.9 版被加入.

curses.**get_tabsize**()

Retrieves the value set by *set_tabsize()*.

在 3.9 版被加入.

curses.**set_tabsize**(*size*)

Sets the number of columns used by the curses library when converting a tab character to spaces as it adds the tab to a window.

在 3.9 版被加入.

curses.**setsyx**(*y*, *x*)

Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both `-1`, then *leaveok* is set `True`.

curses.**setupterm**(*term=None*, *fd=-1*)

Initialize the terminal. *term* is a string giving the terminal name, or `None`; if omitted or `None`, the value of the `TERM` environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or `-1`, the file descriptor for `sys.stdout` will be used.

curses.**start_color**()

Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, *COLORS* and *COLOR_PAIRS*, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()

Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()

Return the value of the environment variable `TERM`, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(*capname*)

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-1` if *capname* is not a Boolean capability, or `0` if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-2` if *capname* is not a numeric capability, or `-1` if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return `None` if *capname* is not a terminfo ”string capability”, or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

curses.**typeahead**(*fd*)

Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.**unctrl**(*ch*)

Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.**ungetch**(*ch*)

Push *ch* so the next *getch()* will return it.

> **備🄵**
>
> Only one *ch* can be pushed before `getch()` is called.

curses.**update_lines_cols**()

Update the *LINES* and *COLS* module variables. Useful for detecting manual screen resize.

在 3.5 版被加入.

curses.**unget_wch**(*ch*)

Push *ch* so the next *get_wch()* will return it.

> **備🄵**
>
> Only one *ch* can be pushed before `get_wch()` is called.

在 3.3 版被加入.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)

Push a *KEY_MOUSE* event onto the input queue, associating the given state data with it.

curses.**use_env**(*flag*)

If used, this function should be called before *initscr()* or newterm are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

curses.**use_default_colors**()

Equivalent to `assume_default_colors(-1, -1)`.

curses.**wrapper**(*func*, */*, **args*, ***kwargs*)

Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to `wrapper()`. Before calling *func*, `wrapper()` turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

17.5.2 Window Objects

class curses.**window**

Window objects, as returned by *initscr()* and *newwin()* above, have the following methods and attributes:

window.**addch**(*ch*[, *attr*])

window.**addch**(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

> **備🄵**
>
> Writing outside the window, subwindow, or pad raises a *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

window.**addnstr**(*str*, *n*[, *attr*])

window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])

Paint at most *n* characters of the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

window.**addstr**(*str*[, *attr*])

window.**addstr**(*y*, *x*, *str*[, *attr*])

Paint the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

> **備🄵**
>
> - Writing outside the window, subwindow, or pad raises *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.
> - A bug in ncurses, the backend for this Python module, can cause SegFaults when resizing windows. This is fixed in ncurses-6.1-20190511. If you are stuck with an earlier ncurses, you can avoid triggering this if you do not call *addstr()* with a *str* that has embedded newlines. Instead, call *addstr()* separately for each line.

window.**attroff**(*attr*)

Remove attribute *attr* from the ”background” set applied to all writes to the current window.

window.**attron**(*attr*)

Add attribute *attr* from the ”background” set applied to all writes to the current window.

window.**attrset**(*attr*)

Set the ”background” set of attributes to *attr*. This set is initially `0` (no attributes).

window.**bkgd**(*ch*[, *attr*])

Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

window.**bkgdset**(*ch*[, *attr*])

Set the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the

blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.`**`border`**`(`[*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]]`)`

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

> **備🄵**
>
> A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

參數	描述	Default value
ls	Left side	*ACS_VLINE*
rs	Right side	*ACS_VLINE*
ts	Top	*ACS_HLINE*
bs	Bottom	*ACS_HLINE*
tl	Upper-left corner	*ACS_ULCORNER*
tr	Upper-right corner	*ACS_URCORNER*
bl	Bottom-left corner	*ACS_LLCORNER*
br	Bottom-right corner	*ACS_LRCORNER*

`window.`**`box`**`(`[*vertch*, *horch*]`)`

Similar to *border()*, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

`window.`**`chgat`**`(`*attr*`)`

`window.`**`chgat`**`(`*num*, *attr*`)`

`window.`**`chgat`**`(`*y*, *x*, *attr*`)`

`window.`**`chgat`**`(`*y*, *x*, *num*, *attr*`)`

Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the *touchline()* method so that the contents will be redisplayed by the next window refresh.

`window.`**`clear`**`()`

Like *erase()*, but also cause the whole window to be repainted upon next call to *refresh()*.

`window.`**`clearok`**`(`*flag*`)`

If *flag* is `True`, the next call to *refresh()* will clear the window completely.

`window.`**`clrtobot`**`()`

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of *clrtoeol()* is performed.

`window.`**`clrtoeol`**`()`

Erase from cursor to the end of the line.

`window.`**`cursyncup`**`()`

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

`window.`**`delch`**`(`[*y*, *x*]`)`

Delete any character at `(y, x)`.

window.**deleteln**()

Delete the line under the cursor. All following lines are moved up by one line.

window.**derwin**(*begin_y*, *begin_x*)

window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

An abbreviation for "derive window", *derwin()* is the same as calling *subwin()*, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.**echochar**(*ch*[, *attr*])

Add character *ch* with attribute *attr*, and immediately call *refresh()* on the window.

window.**enclose**(*y*, *x*)

Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

在 3.10 版的變更: Previously it returned `1` or `0` instead of `True` or `False`.

window.**encoding**

Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with *window.subwin()*. By default, current locale encoding is used (see *locale.getencoding()*).

在 3.3 版被加入.

window.**erase**()

Clear the window.

window.**getbegyx**()

Return a tuple `(y, x)` of coordinates of upper-left corner.

window.**getbkgd**()

Return the given window's current background character/attribute pair.

window.**getch**([*y*, *x*])

Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

window.**get_wch**([*y*, *x*])

Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

在 3.3 版被加入.

window.**getkey**([*y*, *x*])

Get a character, returning a string instead of an integer, as *getch()* does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.**getmaxyx**()

Return a tuple `(y, x)` of the height and width of the window.

window.**getparyx**()

Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

window.**getstr**()

window.**getstr**(*n*)

window.**getstr**(*y*, *x*)

window.**getstr**(*y*, *x*, *n*)

Read a bytes object from the user, with primitive line editing capacity. The maximum value for *n* is 2047.

在 3.14 版的變更: The maximum value for *n* was increased from 1023 to 2047.

window.**getyx**()

Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.**hline**(*ch*, *n*)

window.**hline**(*y*, *x*, *ch*, *n*)

Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.**idcok**(*flag*)

If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*flag*)

If *flag* is `True`, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)

If *flag* is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**(*y*, *x*)

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*, *attr*)

window.**insch**(*y*, *x*, *ch*, *attr*)

Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)

Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()

Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*, *attr*)

window.**insnstr**(*y*, *x*, *str*, *n*, *attr*)

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*, *attr*)

window.**insstr**(*y*, *x*, *str*, *attr*)

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**instr**(*n*)

window.**instr**(*y*, *x*, *n*)

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL). The maximum value for *n* is 2047.

在 3.14 版的變更: The maximum value for *n* was increased from 1023 to 2047.

window.**is_linetouched**(*line*)

Return `True` if the specified line was modified since the last call to *refresh()*; otherwise return `False`. Raise a *curses.error* exception if *line* is not valid for the given window.

window.**is_wintouched**()

Return `True` if the specified window was modified since the last call to *refresh()*; otherwise return `False`.

window.**keypad**(*flag*)

If *flag* is `True`, escape sequences generated by some keys (keypad, function keys) will be interpreted by *curses*. If *flag* is `False`, escape sequences will be left as is in the input stream.

window.**leaveok**(*flag*)

If *flag* is `True`, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is `False`, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)

Move cursor to `(new_y, new_x)`.

window.**mvderwin**(*y*, *x*)

Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)

Move the window so its upper-left corner is at `(new_y, new_x)`.

window.**nodelay**(*flag*)

If *flag* is `True`, *getch()* will be non-blocking.

window.**notimeout**(*flag*)

If *flag* is `True`, escape sequences will not be timed out.

If *flag* is `False`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call *doupdate()*.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overlay()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overwrite()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(*file*)

Write all data associated with the window into the provided file object. This information can be later retrieved using the *getwin()* function.

window.**redrawln**(*beg*, *num*)

Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next `refresh()` call.

window.**redrawwin**()

Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

window.**resize**(*nlines*, *ncols*)

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

window.**scroll**([*lines=1*])

Scroll the screen or scrolling region upward by *lines* lines.

window.**scrollok**(*flag*)

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

window.**setscrreg**(*top*, *bottom*)

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window.**standend**()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.**standout**()

Turn on attribute *A_STANDOUT*.

window.**subpad**(*begin_y*, *begin_x*)

window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

window.**subwin**(*begin_y*, *begin_x*)

window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.**syncdown**()

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

window.**syncok**(*flag*)

If *flag* is `True`, then `syncup()` is called automatically whenever there is a change in the window.

window.**syncup**()

Touch all locations in ancestors of the window that have been changed in the window.

window.**timeout**(*delay*)

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and *getch()* will return -1 if no input is waiting. If *delay* is positive, then *getch()* will block for *delay* milliseconds, and return -1 if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=True) or unchanged (*changed*=False).

window.**touchwin**()

Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()

Mark all lines in the window as unchanged since the last call to *refresh()*.

window.**vline**(*ch*, *n*[, *attr*])

window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])

Display a vertical line starting at (y, x) with length *n* consisting of the character *ch* with attributes *attr*.

17.5.3 Constants

curses 模組定義了以下資料成員：

curses.**ERR**

Some curses routines that return an integer, such as *getch()*, return *ERR* upon failure.

curses.**OK**

Some curses routines that return an integer, such as *napms()*, return *OK* upon success.

curses.**version**

curses.**__version__**

A bytes object representing the current version of the module.

curses.**ncurses_version**

A named tuple containing the three components of the ncurses library version: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so curses.ncurses_version[0] is equivalent to curses.ncurses_version.major and so on.

Availability: if the ncurses library is used.

在 3.8 版被加入.

curses.**COLORS**

The maximum number of colors the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLOR_PAIRS**

The maximum number of color pairs the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLS**

The width of the screen, i.e., the number of columns. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

curses.**LINES**

The height of the screen, i.e., the number of lines. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribute	含義
curses.**A_ALTCHARSET**	Alternate character set mode
curses.**A_BLINK**	Blink mode
curses.**A_BOLD**	Bold mode
curses.**A_DIM**	Dim mode
curses.**A_INVIS**	Invisible or blank mode
curses.**A_ITALIC**	Italic mode
curses.**A_NORMAL**	Normal attribute
curses.**A_PROTECT**	Protected mode
curses.**A_REVERSE**	Reverse background and foreground colors
curses.**A_STANDOUT**	Standout mode
curses.**A_UNDERLINE**	Underline mode
curses.**A_HORIZONTAL**	Horizontal highlight
curses.**A_LEFT**	Left highlight
curses.**A_LOW**	Low highlight
curses.**A_RIGHT**	Right highlight
curses.**A_TOP**	Top highlight
curses.**A_VERTICAL**	Vertical highlight

在 3.7 版被加入: A_ITALIC was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	含義
curses.**A_ATTRIBUTES**	Bit-mask to extract attributes
curses.**A_CHARTEXT**	Bit-mask to extract a character
curses.**A_COLOR**	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Key
curses.**KEY_MIN**	Minimum key value
curses.**KEY_BREAK**	Break key (unreliable)
curses.**KEY_DOWN**	Down-arrow
curses.**KEY_UP**	Up-arrow
curses.**KEY_LEFT**	Left-arrow
curses.**KEY_RIGHT**	Right-arrow
curses.**KEY_HOME**	Home key (upward+left arrow)
curses.**KEY_BACKSPACE**	Backspace (unreliable)
curses.**KEY_F0**	Function keys. Up to 64 function keys are supported.
curses.**KEY_Fn**	Value of function key *n*
curses.**KEY_DL**	Delete line
curses.**KEY_IL**	Insert line
curses.**KEY_DC**	Delete character

繼續下一頁

表格 1 – 繼續上一頁

Key constant	Key
curses.**KEY_IC**	Insert char or enter insert mode
curses.**KEY_EIC**	Exit insert char mode
curses.**KEY_CLEAR**	Clear screen
curses.**KEY_EOS**	Clear to end of screen
curses.**KEY_EOL**	Clear to end of line
curses.**KEY_SF**	Scroll 1 line forward
curses.**KEY_SR**	Scroll 1 line backward (reverse)
curses.**KEY_NPAGE**	Next page
curses.**KEY_PPAGE**	Previous page
curses.**KEY_STAB**	Set tab
curses.**KEY_CTAB**	Clear tab
curses.**KEY_CATAB**	Clear all tabs
curses.**KEY_ENTER**	Enter or send (unreliable)
curses.**KEY_SRESET**	Soft (partial) reset (unreliable)
curses.**KEY_RESET**	Reset or hard reset (unreliable)
curses.**KEY_PRINT**	Print
curses.**KEY_LL**	Home down or bottom (lower left)
curses.**KEY_A1**	Upper left of keypad

繼續下一頁

表格 1 – 繼續上一頁

Key constant	Key
curses.**KEY_A3**	Upper right of keypad
curses.**KEY_B2**	Center of keypad
curses.**KEY_C1**	Lower left of keypad
curses.**KEY_C3**	Lower right of keypad
curses.**KEY_BTAB**	Back tab
curses.**KEY_BEG**	Beg (beginning)
curses.**KEY_CANCEL**	Cancel
curses.**KEY_CLOSE**	Close
curses.**KEY_COMMAND**	Cmd (command)
curses.**KEY_COPY**	Copy
curses.**KEY_CREATE**	Create
curses.**KEY_END**	End
curses.**KEY_EXIT**	Exit
curses.**KEY_FIND**	Find
curses.**KEY_HELP**	Help
curses.**KEY_MARK**	Mark
curses.**KEY_MESSAGE**	Message
curses.**KEY_MOVE**	Move

繼續下一頁

表格 1 – 繼續上一頁

Key constant	Key
curses.**KEY_NEXT**	Next
curses.**KEY_OPEN**	Open
curses.**KEY_OPTIONS**	Options
curses.**KEY_PREVIOUS**	Prev (previous)
curses.**KEY_REDO**	Redo
curses.**KEY_REFERENCE**	Ref (reference)
curses.**KEY_REFRESH**	Refresh
curses.**KEY_REPLACE**	Replace
curses.**KEY_RESTART**	Restart
curses.**KEY_RESUME**	Resume
curses.**KEY_SAVE**	Save
curses.**KEY_SBEG**	Shifted Beg (beginning)
curses.**KEY_SCANCEL**	Shifted Cancel
curses.**KEY_SCOMMAND**	Shifted Command
curses.**KEY_SCOPY**	Shifted Copy
curses.**KEY_SCREATE**	Shifted Create
curses.**KEY_SDC**	Shifted Delete char
curses.**KEY_SDL**	Shifted Delete line

繼續下一頁

表格 1 – 繼續上一頁

Key constant	Key
curses.**KEY_SELECT**	Select
curses.**KEY_SEND**	Shifted End
curses.**KEY_SEOL**	Shifted Clear line
curses.**KEY_SEXIT**	Shifted Exit
curses.**KEY_SFIND**	Shifted Find
curses.**KEY_SHELP**	Shifted Help
curses.**KEY_SHOME**	Shifted Home
curses.**KEY_SIC**	Shifted Input
curses.**KEY_SLEFT**	Shifted Left arrow
curses.**KEY_SMESSAGE**	Shifted Message
curses.**KEY_SMOVE**	Shifted Move
curses.**KEY_SNEXT**	Shifted Next
curses.**KEY_SOPTIONS**	Shifted Options
curses.**KEY_SPREVIOUS**	Shifted Prev
curses.**KEY_SPRINT**	Shifted Print
curses.**KEY_SREDO**	Shifted Redo
curses.**KEY_SREPLACE**	Shifted Replace
curses.**KEY_SRIGHT**	Shifted Right arrow

繼續下一頁

表格 1 – 繼續上一頁

Key constant	Key
curses.**KEY_SRSUME**	Shifted Resume
curses.**KEY_SSAVE**	Shifted Save
curses.**KEY_SSUSPEND**	Shifted Suspend
curses.**KEY_SUNDO**	Shifted Undo
curses.**KEY_SUSPEND**	Suspend
curses.**KEY_UNDO**	Undo
curses.**KEY_MOUSE**	Mouse event has occurred
curses.**KEY_RESIZE**	Terminal resize event
curses.**KEY_MAX**	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (*KEY_F1*, *KEY_F2*, *KEY_F3*, *KEY_F4*) available, and the arrow keys mapped to *KEY_UP*, *KEY_DOWN*, *KEY_LEFT* and *KEY_RIGHT* in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

> **備🄵**
>
> These are available only after *initscr()* has been called.

ACS code	含義
curses.**ACS_BBSS**	alternate name for upper right corner
curses.**ACS_BLOCK**	solid square block
curses.**ACS_BOARD**	board of squares
curses.**ACS_BSBS**	alternate name for horizontal line
curses.**ACS_BSSB**	alternate name for upper left corner
curses.**ACS_BSSS**	alternate name for top tee
curses.**ACS_BTEE**	bottom tee
curses.**ACS_BULLET**	bullet
curses.**ACS_CKBOARD**	checker board (stipple)
curses.**ACS_DARROW**	arrow pointing down
curses.**ACS_DEGREE**	degree symbol
curses.**ACS_DIAMOND**	diamond
curses.**ACS_GEQUAL**	greater-than-or-equal-to
curses.**ACS_HLINE**	horizontal line
curses.**ACS_LANTERN**	lantern symbol
curses.**ACS_LARROW**	left arrow
curses.**ACS_LEQUAL**	less-than-or-equal-to
curses.**ACS_LLCORNER**	lower left-hand corner

繼續下一頁

表格 2 – 繼續上一頁

ACS code	含義
curses.**ACS_LRCORNER**	lower right-hand corner
curses.**ACS_LTEE**	left tee
curses.**ACS_NEQUAL**	not-equal sign
curses.**ACS_PI**	letter pi
curses.**ACS_PLMINUS**	plus-or-minus sign
curses.**ACS_PLUS**	big plus sign
curses.**ACS_RARROW**	right arrow
curses.**ACS_RTEE**	right tee
curses.**ACS_S1**	scan line 1
curses.**ACS_S3**	scan line 3
curses.**ACS_S7**	scan line 7
curses.**ACS_S9**	scan line 9
curses.**ACS_SBBS**	alternate name for lower right corner
curses.**ACS_SBSB**	alternate name for vertical line
curses.**ACS_SBSS**	alternate name for right tee
curses.**ACS_SSBB**	alternate name for lower left corner
curses.**ACS_SSBS**	alternate name for bottom tee
curses.**ACS_SSSB**	alternate name for left tee

繼續下一頁

表格 2 – 繼續上一頁

ACS code	含義
curses.**ACS_SSSS**	alternate name for crossover or big plus
curses.**ACS_STERLING**	pound sterling
curses.**ACS_TTEE**	top tee
curses.**ACS_UARROW**	up arrow
curses.**ACS_ULCORNER**	upper left corner
curses.**ACS_URCORNER**	upper right corner
curses.**ACS_VLINE**	vertical line

The following table lists mouse button constants used by *getmouse()*:

Mouse button constant	含義
curses.**BUTTONn_PRESSED**	Mouse button *n* pressed
curses.**BUTTONn_RELEASED**	Mouse button *n* released
curses.**BUTTONn_CLICKED**	Mouse button *n* clicked
curses.**BUTTONn_DOUBLE_CLICKED**	Mouse button *n* double clicked
curses.**BUTTONn_TRIPLE_CLICKED**	Mouse button *n* triple clicked
curses.**BUTTON_SHIFT**	Shift was down during button state change
curses.**BUTTON_CTRL**	Control was down during button state change
curses.**BUTTON_ALT**	Control was down during button state change

在 3.10 版的變更: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

The following table lists the predefined colors:

Constant	🄵色
curses.**COLOR_BLACK**	黑
curses.**COLOR_BLUE**	藍
curses.**COLOR_CYAN**	Cyan (light greenish blue)
curses.**COLOR_GREEN**	🄵
curses.**COLOR_MAGENTA**	Magenta (purplish red)
curses.**COLOR_RED**	紅
curses.**COLOR_WHITE**	白
curses.**COLOR_YELLOW**	Yellow

17.6 curses.textpad --- Text input widget for curses programs

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

curses.textpad 模組定義了以下函式：

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

17.6.1 Textbox objects

You can instantiate a *Textbox* object as follows:

class curses.textpad.**Textbox**(*win*)

Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods:

edit([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke

entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

do_command(*ch*)

Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

17.7 curses.ascii --- ASCII 字元的工具程式

原始碼： Lib/curses/ascii.py

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

名懲	含義
curses.ascii.**NUL**	
curses.ascii.**SOH**	Start of heading, console interrupt
curses.ascii.**STX**	Start of text
curses.ascii.**ETX**	End of text
curses.ascii.**EOT**	End of transmission
curses.ascii.**ENQ**	Enquiry, goes with *ACK* flow control
curses.ascii.**ACK**	Acknowledgement
curses.ascii.**BEL**	Bell
curses.ascii.**BS**	Backspace
curses.ascii.**TAB**	Tab
curses.ascii.**HT**	Alias for *TAB*: ”Horizontal tab”
curses.ascii.**LF**	Line feed
curses.ascii.**NL**	Alias for *LF*: ”New line”
curses.ascii.**VT**	Vertical tab
curses.ascii.**FF**	Form feed
curses.ascii.**CR**	Carriage return
curses.ascii.**SO**	Shift-out, begin alternate character set
curses.ascii.**SI**	Shift-in, resume default character set

繼續下一頁

表格 3 – 繼續上一頁

名懲	含義
curses.ascii.**DLE**	Data-link escape
curses.ascii.**DC1**	XON, for flow control
curses.ascii.**DC2**	Device control 2, block-mode flow control
curses.ascii.**DC3**	XOFF, for flow control
curses.ascii.**DC4**	Device control 4
curses.ascii.**NAK**	Negative acknowledgement
curses.ascii.**SYN**	Synchronous idle
curses.ascii.**ETB**	End transmission block
curses.ascii.**CAN**	Cancel
curses.ascii.**EM**	End of medium
curses.ascii.**SUB**	Substitute
curses.ascii.**ESC**	Escape
curses.ascii.**FS**	File separator
curses.ascii.**GS**	Group separator
curses.ascii.**RS**	Record separator, block-mode terminator
curses.ascii.**US**	Unit separator
curses.ascii.**SP**	Space
curses.ascii.**DEL**	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter

conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.**isalnum**(*c*)

Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.**isalpha**(*c*)

Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.**isascii**(*c*)

Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.**isblank**(*c*)

Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.**iscntrl**(*c*)

Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.**isdigit**(*c*)

Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.**isgraph**(*c*)

Checks for ASCII any printable character except space.

curses.ascii.**islower**(*c*)

Checks for an ASCII lower-case character.

curses.ascii.**isprint**(*c*)

Checks for any ASCII printable character including space.

curses.ascii.**ispunct**(*c*)

Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.**isspace**(*c*)

Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.**isupper**(*c*)

Checks for an ASCII uppercase letter.

curses.ascii.**isxdigit**(*c*)

Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.**isctrl**(*c*)

Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.**ismeta**(*c*)

Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function *ord()*.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.**ascii**(*c*)

Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.**ctrl**(*c*)

Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.**alt**(*c*)

Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.**unctrl**(*c*)

Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00--0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.**controlnames**

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

17.8 `curses.panel` --- curses 的面板堆🄵擴充

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

17.8.1 函式

curses.panel 模組定義了以下函式：

curses.panel.**bottom_panel**()

Returns the bottom panel in the panel stack.

curses.panel.**new_panel**(*win*)

Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

curses.panel.**top_panel**()

Returns the top panel in the panel stack.

curses.panel.**update_panels**()

Updates the virtual screen after changes in the panel stack. This does not call *curses.doupdate()*, so you'll have to do this yourself.

17.8.2 Panel Objects

Panel objects, as returned by *new_panel()* above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

Panel.**above**()

Returns the panel above the current panel.

Panel.**below**()

Returns the panel below the current panel.

Panel.**bottom**()

Push the panel to the bottom of the stack.

Panel.**hidden**()

Returns `True` if the panel is hidden (not visible), `False` otherwise.

Panel.**hide**()

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.**move**(*y*, *x*)

Move the panel to the screen coordinates `(y, x)`.

Panel.**replace**(*win*)

Change the window associated with the panel to the window *win*.

Panel.**set_userptr**(*obj*)

Set the panel's user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.**show**()

Display the panel (which might have been hidden).

Panel.**top**()

Push panel to the top of the stack.

Panel.**userptr**()

Returns the user pointer for the panel. This might be any Python object.

Panel.**window**()

Returns the window object associated with the panel.

17.9 cmd --- 以列🄵導向的命令直譯器支援

原始碼： Lib/cmd.py

Cmd 類🄵提供了一個簡單的架構，用於撰寫列導向的命令直譯器。這類直譯器常用於測試控制工具、管理工具以及日後將包裝於更高階介面的原型中。

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

Cmd 實例或其子類🄵實例是一種列導向的直譯器架構。通常🄵有必要直接實例化*Cmd* 本身；它更適合作🄵你自定義的直譯器類🄵的父類🄵，讓你能繼承*Cmd* 的方法，🄵封裝動作方法。

可選引數 *completekey* 是*readline* 模組中用於自動完成的按鍵名稱；預設🄵 `Tab`。若 *completekey* 不🄵*None* 且*readline* 可用，則會自動🄴用命令自動完成功能。

預設值 `'tab'` 會被特殊處理，使其在所有的*readline.backend* 中皆代表 `Tab` 鍵。具體來🄵，若*readline.backend* 🄵 `editline`，則 `Cmd` 會改用 `'^I'` 取代 `'tab'`。請注意，其他值不會有此處理方式，且可能僅能在特定的後端中適用。

可選引數 *stdin* 與 *stdout* 用來指定 Cmd 實例或其子類🄵實例所使用的輸入與輸出檔案物件。若未指定，預設🄵*sys.stdin* 與*sys.stdout*。

若你希望使用指定的 *stdin*，請務必將該實例的*use_rawinput* 屬性設🄵 `False`，否則 *stdin* 會被忽略。

在 3.13 版的變更: 對於 `editline`，`completekey='tab'` 會被替🄵🄵 `'^I'`。

17.9.1 Cmd 物件

Cmd 實例具有以下方法：

Cmd.**cmdloop**(*intro=None*)

重🄵顯示提示字元、接收輸入、剖析接收到的輸入字串前綴，🄵派發給動作方法，將其余部分作🄵引數傳遞給它們

此可選引數🄵🄵幅或導言字串，會在首次顯示提示字元前輸出（此值會覆寫*intro* 類🄵屬性）。

如果已載入*readline* 模組，輸入將自動繼承類似 **bash** 的歷史紀🄵編輯功能（例如 Control-P 可向上🄵動至上一個命令，Control-N 向下🄵動至下一個命令，Control-F 非破壞性地將游標向右移動，Control-B 非破壞性地將游標向左移動等）。

當輸入🄵檔案結尾（EOF）時，會傳回字串 'EOF'。

直譯器實例僅當存在 do_foo() 方法時，才會識🄵命令名稱 foo。作🄵特殊情🄵，以字元 '?' 開頭的列會被派發至*do_help()* 方法；另一個特殊情🄵是，以字元 '!' 開頭的列會被派發至 do_shell() 方法（若該方法已定義）。

當*postcmd()* 方法回傳真值時，此方法將會結束。傳遞給*postcmd()* 的 *stop* 引數是該命令對應的 do_*() 方法的回傳值。

如果🄵用了自動完成，命令的自動完成將會自動執行，而命令引數的自動完成則是透過呼叫 complete_foo() 方法🄵傳入 *text*、*line*、*begidx* 和 *endidx* 引數來處理。*text* 是要比對的字串前綴：所有回傳的符合項都必須以此字串開頭。*line* 是目前的輸入列（前置空白會被移除），*begidx* 和 *endidx* 則分🄵是前綴字串的起始與結束索引，可用來根據引數所在的位置提供不同的自動完成結果。

Cmd.**do_help**(*arg*)

所有*Cmd* 的子類🄵都會繼承預先定義的 do_help() 方法。當此方法接收到引數 'bar' 時，會呼叫對應的 help_bar() 方法；若該方法不存在，則會列印 do_bar() 的🄵明字串（若有的話）。若未提供任何引數，do_help() 會列出所有可用的🄵明主題（也就是所有具有對應 help_*() 方法或有🄵明字串的命令），🄵且也會列出所有尚未記🄵的命令。

Cmd.**onecmd**(*str*)

將引數視🄵在回應提示字元時所輸入的🄵容來直譯。這個方法可以被覆寫，但通常不需要這🄵做；參見*precmd()* 與*postcmd()* 方法，它們提供實用的執行勾點（hook）。此方法的回傳值是一個旗標，用來指出是否應該停止直譯器對命令的直譯。若有對應 *str* 命令的 do_*() 方法，則會回傳該方法的回傳值；否則，回傳值將來自*default()* 方法。

Cmd.**emptyline**()

在回應提示字元時輸入空白列，會呼叫此方法。若此方法未被覆寫，則會重復上一次輸入的非空命令。

Cmd.**default**(*line*)

當輸入列中的命令前綴無法辨識時，會呼叫此方法。若此方法未被覆寫，則會輸出🄵回傳錯誤訊息。

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)

當🄵有對應特定命令的 complete_*() 方法時，會呼叫此方法以完成輸入列。預設會回傳空串列。

Cmd.**columnize**(*list*, *displaywidth=80*)

此方法用來將字串串列顯示🄵緊🄵的欄集合。每一欄的寬度僅足以容納其🄵容，各欄之間以兩個空格分隔，以提高可讀性。

Cmd.**precmd**(*line*)

勾點方法會在直譯命令列 *line* 前執行，但會在提示字元🄵生🄵顯示後才觸發。這個方法在*Cmd* 類🄵中🄵 stub，預期由子類🄵覆寫。回傳值會作🄵*onecmd()* 方法所執行的命令；*precmd()* 的實作可以重寫該命令，或直接回傳未變更的 *line*。

Cmd.**postcmd**(*stop*, *line*)

勾點方法會在命令派發完成後執行。這個方法在*Cmd* 類🄵中🄵 stub，預期由子類🄵覆寫。*line* 是剛剛執行的命令列，而 *stop* 是一個旗標，用來指出在呼叫*postcmd()* 後是否應終止執行；該值即

🄵*onecmd()* 方法的回傳值。本方法的回傳值將會更新🄵部的 *stop* 旗標；若回傳 false，則會繼續進行直譯。

Cmd.**preloop**()

當*cmdloop()* 被呼叫時，此勾點方法會執行一次。這個方法在*Cmd* 類🄵中🄵 stub，預期由子類🄵覆寫。

Cmd.**postloop**()

當*cmdloop()* 即將回傳時，此勾點方法會執行一次。這個方法在*Cmd* 類🄵中🄵 stub，預期由子類🄵覆寫。

Cmd 子類🄵的實例包含一些公開的實例變數:

Cmd.**prompt**

用來請求輸入的提示字元。

Cmd.**identchars**

可作🄵命令前綴的字元字串

Cmd.**lastcmd**

最後一個遇到的非空命令前綴

Cmd.**cmdqueue**

排入🄵列的輸入列串列。當*cmdloop()* 需要新輸入時，會檢查 cmdqueue 串列；若不🄵空，其元素將依序處理，就如同它們是在提示字元中輸入的一樣。

Cmd.**intro**

作🄵簡介或🄵幅的字串。可透過🄵*cmdloop()* 方法提供引數來覆寫此🄵容。

Cmd.**doc_header**

若🄵明輸出包含已記🄵命令的區段，則會顯示的標頭字串。

Cmd.**misc_header**

若🄵明輸出包含雜項🄵明主題的區段（也就是存在 `help_*()` 方法但🄵有對應的 `do_*()` 方法），則會顯示的標頭字串。

Cmd.**undoc_header**

若🄵明輸出包含未記🄵命令的區段（也就是存在 `do_*()` 方法但🄵有對應的 `help_*()` 方法），則會顯示的標頭字串。

Cmd.**ruler**

用於在🄵明訊息的標頭下方繪🄵分隔🄵的字元。若🄵空，則不會繪🄵分隔🄵。預設🄵 `'='`。

Cmd.**use_rawinput**

一個旗標，預設🄵 true。若🄵 true，*cmdloop()* 會使用*input()* 來顯示提示字元🄵讀取下一個命令；若🄵 false，則會改用*sys.stdout.write()* 和*sys.stdin.readline()*。（這表示在支援的系統中，透過 import *readline* module，直譯器將自動支援類似 **Emacs** 的列編輯與命令歷史快捷鍵。）

17.9.2 Cmd 范例

cmd module 主要用於建構自訂 shell，讓使用者能以互動方式操作程式。

本節將示範如何以*turtle* module 中的幾個命令🄵基礎，建立一個簡單的 shell。

像是*forward()* 這樣的基本 turtle 命令，可透過新增名🄵 `do_forward()` 的方法加入至*Cmd* 子類🄵中。傳入的引數會轉🄵🄵數值，🄵傳送給 turtle 模組。該方法的🄵明字串會用於 shell 所提供的🄵明功能中。

此範例同時包含一個簡單的🄵🄵與重播功能，其實作方式是透過*precmd()* 方法，負責將輸入轉🄵小寫🄵寫入檔案。`do_playback()` 方法則會讀取該檔案，🄵將🄵🄵的命令加入*cmdqueue* 中以供立即重播:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = '歡迎來到 turtle shell。輸入 help 或 ? 來列出命令。\n'
    prompt = '(turtle) '
    file = None

    # ----- 基本烏龜命令 -----
    def do_forward(self, arg):
        '將烏龜向前移動指定的距離: FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        '將烏龜右轉指定的角度: RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        '將烏龜左轉指定的角度: LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        '將烏龜移動到指定的🄵對位置🄵改變方向。 GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        '將烏龜返回起始位置: HOME'
        home()
    def do_circle(self, arg):
        '畫出指定半徑、範圍和步數的圓: CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        '顯示當前烏龜位置: POSITION'
        print('當前位置是 %d %d\n' % position())
    def do_heading(self, arg):
        '顯示當前烏龜方向角度: HEADING'
        print('當前方向是 %d\n' % (heading(),))
    def do_color(self, arg):
        '設定🄵色: COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        '撤銷 (重🄵) 最後一次烏龜動作: UNDO'
    def do_reset(self, arg):
        '清除畫面🄵將烏龜返回到中心: RESET'
        reset()
    def do_bye(self, arg):
        '停止🄵🄵、關閉烏龜視窗🄵退出: BYE'
        print('感謝使用 Turtle')
        self.close()
        bye()
        return True

    # ----- 🄵🄵與回放 -----
    def do_record(self, arg):
        '將未來命令儲存至檔案: RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        '從檔案回放命令: PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
```

(繼續下一頁)

(繼續上一頁)

```
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    '將一串零個或多個數字轉🄵🄵引數 tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

以下是與 turtle shell 的一段範例互動，展示了🄵明功能、使用空白列重🄵執行命令，以及簡單的🄵🄵與重播功能：

```
歡迎來到 turtle shell。輸入 help 或 ? 來列出命令。

(turtle) ?

已記🄵的命令（輸入 help <主題>）：
=========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
將烏龜向前移動指定的距離： FORWARD 10
(turtle) record spiral.cmd
(turtle) position
當前位置是 0 0

(turtle) heading
當前方向是 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
當前方向是 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
```

(繼續下一頁)

(繼續上一頁)

```
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
當前位置是 0 0

當前方向是 0

當前方向是 180

(turtle) bye
感謝使用 Turtle
```

CHAPTER 18

🄵行執行 (Concurrent Execution)

本章節描述的模組在程式的🄵行執行上提供支援。選擇要使用哪一個工具則取🄵於是執行什🄵樣的任務（CPU 密集或 IO 密集）與偏好的開發風格（事件驅動協作式多工處理或搶占式多工處理）。以下🄵此章節總覽：

18.1 `threading` --- 基於執行緒的平行性

原始碼： Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level *`_thread`* module.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

18.1.1 Introduction

The `threading` module provides a way to run multiple threads (smaller units of a process) concurrently within a single process. It allows for the creation and management of threads, making it possible to execute tasks in parallel, sharing memory space. Threads are particularly useful when tasks are I/O bound, such as file operations or making network requests, where much of the time is spent waiting for external resources.

A typical use case for `threading` includes managing a pool of worker threads that can process multiple tasks concurrently. Here's a basic example of creating and starting threads using *`Thread`*:

```
import threading
import time

def crawl(link, delay=3):
    print(f"crawl started for {link}")
    time.sleep(delay)  # Blocking I/O (simulating a network request)
    print(f"crawl ended for {link}")

links = [
    "https://python.org",
    "https://docs.python.org",
    "https://peps.python.org",
```

(繼續下一頁)

(繼續上一頁)

```
]

# Start threads for each link
threads = []
for link in links:
    # Using `args` to pass positional arguments and `kwargs` for keyword arguments
    t = threading.Thread(target=crawl, args=(link,), kwargs={"delay": 2})
    threads.append(t)

# Start each thread
for t in threads:
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()
```

在 3.7 版的變更: This module used to be optional, it is now always available.

也參考

concurrent.futures.ThreadPoolExecutor offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

queue provides a thread-safe interface for exchanging data between running threads.

asyncio offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

備🄵

In the Python 2.x series, this module contained `camelCase` names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower.

CPython 實作細節： In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *multiprocessing* or *concurrent.futures.ProcessPoolExecutor*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

18.1.2 GIL and performance considerations

Unlike the *multiprocessing* module, which uses separate processes to bypass the *global interpreter lock* (GIL), the threading module operates within a single process, meaning that all threads share the same memory space. However, the GIL limits the performance gains of threading when it comes to CPU-bound tasks, as only one thread can execute Python bytecode at a time. Despite this, threads remain a useful tool for achieving concurrency in many scenarios.

As of Python 3.13, *free-threaded* builds can disable the GIL, enabling true parallel execution of threads, but this feature is not available by default (see **PEP 703**).

18.1.3 Reference

此模組定義了以下函式：

`threading.`**`active_count`**`()`

Return the number of *Thread* objects currently alive. The returned count is equal to the length of the list returned by *enumerate()*.

The function `activeCount` is a deprecated alias for this function.

`threading.`**`current_thread`**`()`

Return the current *Thread* object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

`currentThread` 函式是此函式的已🄵用🄵名。

`threading.`**`excepthook`**`(`*args*, /`)`

處理由*Thread.run()* 引發且未被捕捉到的例外。

args 引數有下列屬性：

- *exc_type*：例外型🄵。
- *exc_value*: Exception value, can be `None`.
- *exc_traceback*: Exception traceback, can be `None`.
- *thread*：引發例外的執行緒，可以是 `None`。

If *exc_type* is *SystemExit*, the exception is silently ignored. Otherwise, the exception is printed out on *sys.stderr*.

如果這個函式引發例外，則會呼叫*sys.excepthook()* 來處理它。

threading.excepthook() can be overridden to control how uncaught exceptions raised by *Thread.run()* are handled.

Storing *exc_value* using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing *thread* using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing *thread* after the custom hook completes to avoid resurrecting objects.

也參考

sys.excepthook() 處理未被捕捉到的例外。

在 3.8 版被加入.

`threading.`**`__excepthook__`**

Holds the original value of *threading.excepthook()*. It is saved so that the original value can be restored in case they happen to get replaced with broken or alternative objects.

在 3.10 版被加入.

`threading.`**`get_ident`**`()`

Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

在 3.3 版被加入.

`threading.`**`get_native_id`**`()`

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

可用性: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

在 3.8 版被加入.

在 3.13 版的變更: 新增對 GNU/kFreeBSD 的支援。

threading.**enumerate**()

Return a list of all *Thread* objects currently active. The list includes daemonic threads and dummy thread objects created by *current_thread()*. It excludes terminated threads and threads that have not yet been started. However, the main thread is always part of the result, even when terminated.

threading.**main_thread**()

Return the main *Thread* object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

在 3.4 版被加入.

threading.**settrace**(*func*)

Set a trace function for all threads started from the `threading` module. The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

threading.**settrace_all_threads**(*func*)

Set a trace function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

在 3.12 版被加入.

threading.**gettrace**()

Get the trace function as set by *settrace()*.

在 3.10 版被加入.

threading.**setprofile**(*func*)

Set a profile function for all threads started from the `threading` module. The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

threading.**setprofile_all_threads**(*func*)

Set a profile function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

在 3.12 版被加入.

threading.**getprofile**()

Get the profiler function as set by *setprofile()*.

在 3.10 版被加入.

threading.**stack_size**([*size*])

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

可用性: Windows, pthreads.

Unix platforms with POSIX threads support.

此模組也定義了以下常數:

threading.**TIMEOUT_MAX**

The maximum value allowed for the *timeout* parameter of blocking functions (*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, etc.). Specifying a timeout greater than this value will raise an *OverflowError*.

在 3.2 版被加入.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's *Thread* class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

Thread-local data

Thread-local data is data whose values are thread specific. If you have data that you want to be local to a thread, create a *local* object and use its attributes:

```
>>> mydata = local()
>>> mydata.number = 42
>>> mydata.number
42
```

You can also access the *local*-object's dictionary:

```
>>> mydata.__dict__
{'number': 42}
>>> mydata.__dict__.setdefault('widgets', [])
[]
>>> mydata.widgets
[]
```

If we access the data in a different thread:

```
>>> log = []
>>> def f():
...     items = sorted(mydata.__dict__.items())
...     log.append(items)
...     mydata.number = 11
...     log.append(mydata.number)

>>> import threading
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[], 11]
```

we get different data. Furthermore, changes made in the other thread don't affect data seen in this thread:

```
>>> mydata.number
42
```

Of course, values you get from a *local* object, including their `__dict__` attribute, are for whatever thread was current at the time the attribute was read. For that reason, you generally don't want to save these values across threads, as they apply only to the thread they came from.

You can create custom *local* objects by subclassing the *local* class:

```
>>> class MyLocal(local):
...     number = 2
...     def __init__(self, /, **kw):
...         self.__dict__.update(kw)
...     def squared(self):
...         return self.number ** 2
```

This can be useful to support default values, methods and initialization. Note that if you define an `__init__()` method, it will be called each time the *local* object is used in a separate thread. This is necessary to initialize each thread's dictionary.

現在如果我們建立一個*local* 物件：

```
>>> mydata = MyLocal(color='red')
```

會有一個預設的數字：

```
>>> mydata.number
2
```

和一個初始𝔽色：

```
>>> mydata.color
'red'
>>> del mydata.color
```

和一個操作資料的方法：

```
>>> mydata.squared()
4
```

As before, we can access the data in a separate thread:

```
>>> log = []
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[('color', 'red')], 11]
```

without affecting this thread's data:

```
>>> mydata.number
2
>>> mydata.color
Traceback (most recent call last):
...
AttributeError: 'MyLocal' object has no attribute 'color'
```

Note that subclasses can define *__slots__*, but they are not thread local. They are shared across threads:

```
>>> class MyLocal(local):
...     __slots__ = 'number'

>>> mydata = MyLocal()
>>> mydata.number = 42
>>> mydata.color = 'red'
```

So, the separate thread:

```
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
```

affects what we see:

```
>>> mydata.number
11
```

class threading.**local**

A class that represents thread-local data.

Thread 物件

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and *run()* methods of this class.

Once a thread object is created, its activity must be started by calling the thread's *start()* method. This invokes the *run()* method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its *run()* method terminates -- either normally, or by raising an unhandled exception. The *is_alive()* method tests whether the thread is alive.

Other threads can call a thread's *join()* method. This blocks the calling thread until the thread whose *join()* method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the *name* attribute.

If the *run()* method raises an exception, *threading.excepthook()* is called to handle it. By default, *threading.excepthook()* ignores silently *SystemExit*.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the *daemon* property or the *daemon* constructor argument.

> **備🄵**
>
> Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an *Event*.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be *joined*. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*, *context=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

target is the callable object to be invoked by the *run()* method. Defaults to `None`, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number, or "Thread-*N* (target)" where "target" is `target.__name__` if the *target* argument is specified.

args is a list or tuple of arguments for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, *daemon* explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

context is the *`Context`* value to use when starting the thread. The default value is `None` which indicates that the *`sys.flags.thread_inherit_context`* flag controls the behaviour. If the flag is true, threads will start with a copy of the context of the caller of *`start()`*. If false, they will start with an empty context. To explicitly start with an empty context, pass a new instance of *`Context()`*. To explicitly start with a copy of the current context, pass the value from *`copy_context()`*. The flag defaults true on free-threaded builds and false otherwise.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

在 3.3 版的變更: 新增 *daemon* 參數。

在 3.10 版的變更: Use the *target* name if *name* argument is omitted.

在 3.14 版的變更: 新增 *context* 參數。

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's *`run()`* method to be invoked in a separate thread of control.

This method will raise a *`RuntimeError`* if called more than once on the same thread object.

If supported, set the operating system thread name to *`threading.Thread.name`*. The name can be truncated depending on the operating system thread name limits.

在 3.14 版的變更: Set the operating system thread name.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard *`run()`* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with positional and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using list or tuple as the *args* argument which passed to the *`Thread`* could achieve the same effect.

舉例來🄵:

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

Wait until the thread terminates. This blocks the calling thread until the thread whose *`join()`* method is called terminates -- either normally or through an unhandled exception -- or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As *`join()`* always returns `None`, you must call *`is_alive()`* after *`join()`* to decide whether a timeout happened -- if the thread is still alive, the *`join()`* call timed out.

When the *timeout* argument is not present or `None`, the operation will block until the thread terminates.

A thread can be joined many times.

join() raises a *RuntimeError* if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to *join()* a thread before it has been started and attempts to do so raise the same exception.

If an attempt is made to join a running daemonic thread in late stages of *Python finalization* `join()` raises a *PythonFinalizationError*.

在 3.14 版的變更: May raise *PythonFinalizationError*.

name

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

On some platforms, the thread name is set at the operating system level when the thread starts, so that it is visible in task managers. This name may be truncated to fit in a system-specific limit (for example, 15 bytes on Linux or 63 bytes on macOS).

Changes to *name* are only reflected at the OS level when the currently running thread is renamed. (Setting the *name* attribute of a different thread only updates the Python Thread object.)

getName()

setName()

Deprecated getter/setter API for *name*; use it directly as a property instead.

在 3.10 版之後被🄵用.

ident

The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer. See the *get_ident()* function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

native_id

The Thread ID (`TID`) of this thread, as assigned by the OS (kernel). This is a non-negative integer, or `None` if the thread has not been started. See the *get_native_id()* function. This value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

> **備🄵**
>
> Similar to Process IDs, Thread IDs are only valid (guaranteed unique system-wide) from the time the thread is created until the thread has been terminated.

可用性: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

在 3.8 版被加入.

is_alive()

Return whether the thread is alive.

This method returns `True` just before the *run()* method starts until just after the *run()* method terminates. The module function *enumerate()* returns a list of all alive threads.

daemon

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before *start()* is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *daemon* = `False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()

setDaemon()

Deprecated getter/setter API for *daemon*; use it directly as a property instead.

在 3.10 版之後被🄵用.

Lock 物件

原始鎖 (primitive lock) 是一種同步原語 (synchronization primitive)，在鎖定時不屬於特定執行緒。在 Python 中，它是目前可用的最低階同步原語，直接由*_thread* 擴充模組實作。

原始鎖會處於兩種狀態之一：「鎖定 (locked)」或「未鎖定 (unclocked)」，建立時會處於未鎖定狀態。它有兩個基本方法*acquire()* 和*release()*。當狀態🄵未鎖定時，*acquire()* 會將狀態變更🄵鎖定🄵立即回傳。當狀態被鎖定時，*acquire()* 會阻塞 (block)，直到另一個執行緒中對*release()* 的呼叫將其更改🄵未鎖定狀態，然後*acquire()* 呼叫會將其重置🄵鎖定🄵回傳。*release()* 方法只能在鎖定狀態下呼叫；它將狀態更改🄵未鎖定🄵立即回傳。如果嘗試釋放未鎖定的鎖，則會引發*RuntimeError*。

鎖也支援情境管理協定。

當多個執行緒阻塞在*acquire()* 中等待狀態轉變🄵未鎖定，此時若呼叫*release()* 將狀態重置🄵未鎖定，則只會有一個執行緒繼續進行；哪一個等待執行緒會繼續進行是未定義的，🄵且可能因實作而🄵。

所有方法均以最小不可分割的操作方式 (atomically) 執行。

class threading.**Lock**

實作原始鎖物件的類🄵。一旦執行緒獲得了鎖，後續再嘗試獲得它就會被阻塞，直到鎖被釋放；任何執行緒都可以去釋放它。

在 3.13 版的變更: `Lock` 現在是一個類🄵。在早期的 Python 中，`Lock` 是一個會回傳底層私有鎖型🄵實例的工廠函式。

acquire(*blocking=True*, *timeout=-1*)

阻塞或非阻塞地取得鎖。

當以 *blocking* 引數設🄵 `True`（預設值）來叫用，將會阻塞直到鎖被解鎖，然後將其設🄵鎖定🄵回傳 `True`。

當以 *blocking* 引數設🄵 `False` 叫用則不會阻塞。如果 *blocking* 設定🄵 `True` 的呼叫會阻塞，則立即回傳 `False`；否則將鎖設🄵鎖定🄵回傳 `True`。

當使用設定🄵正值的浮點 *timeout* 引數進行叫用，只要持續無法取得鎖，最多會阻塞 *timeout* 指定的秒數。`-1` 的 *timeout* 引數代表指定🄵不會停止的等待。當 *blocking* 🄵 `False` 時禁止指定 *timeout*。

如果成功取得鎖，則回傳值🄵 `True`，否則回傳值🄵 `False`（例如像是 *timeout* 已逾期）。

在 3.2 版的變更: 新的 *timeout* 參數。

在 3.2 版的變更: 如果底層執行緒實作支援的話，鎖的取得現在可以被 POSIX 上的訊號中斷。

在 3.14 版的變更: Lock acquisition can now be interrupted by signals on Windows.

release()

釋放鎖。這可以從任何執行緒呼叫，而不是只有獲得鎖的執行緒。

當鎖被鎖定時，將其重置🄵未鎖定然後回傳。如果任何其他執行緒在等待鎖被解鎖時被阻塞，只允許其中一個執行緒繼續進行。

當在未鎖定的鎖上叫用時，會引發*RuntimeError*

🄵有回傳值。

locked()

如果有取得了鎖，則回傳 `True`。

RLock 物件

可重入鎖 (reentrant lock) 是一種同步原語，同一執行緒可以多次取得它。在🄵部，除了原始鎖使用的鎖定/未鎖定狀態之外，它還使用「所屬執行緒 (owning thread)」和「遞🄵等級 (recursion level)」的概念。在鎖定狀態下，某個執行緒會擁有鎖；在未鎖定狀態下則🄵有執行緒擁有它。

執行緒呼叫鎖的*acquire()* 方法來鎖定它，🄵呼叫它的*release()* 方法來解鎖它。

備🄵

可重入鎖支援*情境管理協定*，因此建議使用 `with` 而不是手動呼叫*acquire()* 和*release()* 來對程式碼區塊處理鎖的獲得和釋放。

RLock 的*acquire()*/*release()* 呼叫成對組合可以嵌套使用，這與 Lock 的*acquire()*/*release()* 不同。只有最後一個*release()*（最外面一對的*release()*）會將鎖重置🄵未鎖定狀態，🄵允許在*acquire()* 中阻塞的另一個執行緒繼續進行。

acquire()/*release()* 必須成對使用：每次獲得都必須在已獲得鎖的執行緒中有一個釋放。如果鎖釋放的次數不能和取得的次數一樣的話，可能會導致死鎖 (deadlock)。

class threading.**RLock**

此類🄵實作了可重入鎖物件。可重入鎖必須由獲得它的執行緒釋放。一旦一個執行緒獲得了可重入鎖，同一個執行緒可以再次獲得它而不會阻塞；執行緒每次獲得它也都必須釋放它一次。

請注意，`RLock` 實際上是一個工廠函式，它會回傳平台有支援的特定 RLock 類🄵的最高效率版本的實例。

acquire(*blocking=True*, *timeout=-1*)

阻塞或非阻塞地取得鎖。

也參考

將 RLock 用作🄴情境管理器
若是使用場景合理，和手動呼叫 `acquire()` 和*release()* 相比，會是更🄵推薦的使用方式。

當以 *blocking* 引數設🄵 `True`（預設值）來呼叫：

- 如果🄵有執行緒擁有鎖，則獲得鎖🄵立即回傳。
- 如果另一個執行緒擁有鎖，則阻塞直到能🄵取得鎖，或者達到 *timeout*（如果設定🄵正浮點值）。
- 如果同一個執行緒擁有鎖，則再次取得鎖，🄵立即回傳。這就是*Lock* 和 `RLock` 之間的差🄵；*Lock* 處理方式與上一種情🄵相同，會阻塞直到能🄵取得鎖。

當以 *blocking* 引數設🄵 `False` 來呼叫：

- 如果🄵有執行緒擁有鎖，則獲得鎖🄵立即回傳。
- 如果另一個執行緒擁有該鎖，則立即回傳。
- 如果同一個執行緒擁有鎖，則再次取得鎖🄵立即回傳。

在所有情🄵下，如果執行緒能🄵取得鎖則回傳 `True`。如果執行緒無法取得鎖（即🄵有阻塞或已達超時限制）則回傳 `False`。

如果多次呼叫，又未能呼叫相同次數的*release()*，則可能會導致死鎖。考慮將 `RLock` 作🄵情境管理器使用，而不是直接呼叫 acquire/release。

在 3.2 版的變更: 新的 *timeout* 參數。

release()

釋放鎖🄵🄵少遞🄵等級。如果被🄵至零，則將鎖重置🄵未鎖定（不屬於任何執行緒），🄵且如果任何其他執行緒被阻塞以等待鎖變成未鎖定狀態，則僅允許其中一個執行緒繼續進行。如果遞🄵後遞🄵等級仍然非零，則鎖會保持鎖定🄵由呼叫它的執行緒所擁有。

僅當呼叫的執行緒擁有鎖時才能呼叫此方法。如果在未取得鎖時呼叫此方法則會引發*RuntimeError*。

🄵有回傳值。

locked()

Return a boolean indicating whether this object is locked right now.

在 3.14 版被加入.

Condition objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the *context management protocol*: using the `with` statement acquires the associated lock for the duration of the enclosed block. The *acquire()* and *release()* methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The *wait()* method releases the lock, and then blocks until another thread awakens it by calling *notify()* or *notify_all()*. Once awakened, *wait()* re-acquires the lock and returns. It is also possible to specify a timeout.

The *notify()* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *notify_all()* method wakes up all threads waiting for the condition variable.

Note: the *notify()* and *notify_all()* methods don't release the lock; this means that the thread or threads awakened will not return from their *wait()* call immediately, but only when the thread that called *notify()* or *notify_all()* finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call *wait()* repeatedly until they see the desired state, while threads that modify the state call *notify()* or *notify_all()* when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because *wait()* can return after an arbitrary long time, and the condition which prompted the *notify()* call may no longer hold true. This is inherent to multi-threaded programming. The *wait_for()* method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between *notify()* and *notify_all()*, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not `None`, it must be a *Lock* or *RLock* object, and it is used as the underlying lock. Otherwise, a new *RLock* object is created and used as the underlying lock.

在 3.3 版的變更: changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

locked()

Return a boolean indicating whether this object is locked right now.

在 3.14 版被加入.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an *RLock*, it is not released using its *release()* method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the *RLock* class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given *timeout* expired, in which case it is `False`.

在 3.2 版的變更: Previously, the method always returned `None`.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call *wait()* repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with *wait()*: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

在 3.2 版被加入.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its *wait()* call until it can reacquire the lock. Since *notify()* does not release the lock, its caller should.

notify_all()

Wake up all threads waiting on this condition. This method acts like *notify()*, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

The method `notifyAll` is a deprecated alias for this method.

Semaphore 物件

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of *acquire()* and *release()*).

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some other thread calls *release()*.

Semaphores also support the *context management protocol*.

class threading.**Semaphore**(*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of *release()* calls minus the number of *acquire()* calls, plus an initial value. The *acquire()* method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, *ValueError* is raised.

在 3.3 版的變更: changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)

Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to *release()*. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to *release()*. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to `False`, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

在 3.2 版的變更: 新的 *timeout* 參數。

release(*n=1*)

Release a semaphore, incrementing the internal counter by *n*. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up *n* of those threads.

在 3.9 版的變更: Added the *n* parameter to release multiple waiting threads at once.

class threading.**BoundedSemaphore**(*value=1*)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, *ValueError* is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

在 3.3 版的變更: changed from a factory function to a class.

Semaphore 范例

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... 使用該連🄵 ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

Event objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true.

class threading.**Event**

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

在 3.3 版的變更: changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

The method `isSet` is a deprecated alias for this method.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call *wait()* once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

wait(*timeout=None*)

Block as long as the internal flag is false and the timeout, if given, has not expired. The return value represents the reason that this blocking method returned; `True` if returning because the internal flag is set to true, or `False` if a timeout is given and the internal flag did not become true within the given wait time.

When the timeout argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds, or fractions thereof.

在 3.1 版的變更: Previously, the method always returned `None`.

Timer 物件

This class represents an action that should be run only after a certain amount of time has passed --- a timer. *Timer* is a subclass of *Thread* and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their *Timer.start* method. The timer can be stopped (before its action has begun) by calling the *cancel()* method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

舉例來🄵:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # 30 秒後會印出 "hello, world"
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

在 3.3 版的變更: changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

Barrier 物件

在 3.2 版被加入.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the *wait()* method and will block until all of the threads have made their *wait()* calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)
```

(繼續下一頁)

(繼續上一頁)

```
def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the *wait()* method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* -- 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # 只會有一個執行緒會印出這個
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the *BrokenBarrierError* exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the threads needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception threading.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

18.1.4 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire` and `release` methods can be used as context managers for a `with` statement. The `acquire` method will be called when the block is entered, and `release` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # 做某些事情...
```

is equivalent to:

```
some_lock.acquire()
try:
    # 做某些事情...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

18.2 `multiprocessing` --- 以行程🄵基礎的平行性

原始碼： Lib/multiprocessing/

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

18.2.1 簡介

multiprocessing is a package that supports spawning processes using an API similar to the *threading* module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

The *multiprocessing* module also introduces APIs which do not have analogs in the *threading* module. A prime example of this is the *Pool* object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```

也參考

concurrent.futures.ProcessPoolExecutor offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the *Pool* interface directly, the

concurrent.futures API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

Process 類🄵

In *multiprocessing*, processes are spawned by creating a *Process* object and then calling its *start()* method. *Process* follows the API of *threading.Thread*. A trivial example of a multiprocess program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why the `if __name__ == '__main__'` part is necessary, see *Programming guidelines*.

The arguments to *Process* usually need to be unpickleable from within the child process. If you tried typing the above example directly into a REPL it could lead to an *AttributeError* in the child process trying to locate the *f* function in the __main__ module.

Contexts and start methods

Depending on the platform, *multiprocessing* supports three ways to start a process. These *start methods* are

spawn
: The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object's *run()* method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

  Available on POSIX and Windows platforms. The default on Windows and macOS.

fork
: The parent process uses *os.fork()* to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

  Available on POSIX systems.

在 3.14 版的變更: This is no longer the default start method on any platform. Code that requires *fork* must explicitly specify that via `get_context()` or `set_start_method()`.

在 3.12 版的變更: If Python is able to detect that your process has multiple threads, the `os.fork()` function that this start method calls internally will raise a `DeprecationWarning`. Use a different start method. See the `os.fork()` documentation for further explanation.

forkserver

When the program starts and selects the *forkserver* start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux. The default on those.

在 3.14 版的變更: This became the default start method on POSIX platforms.

在 3.4 版的變更: *spawn* added on all POSIX platforms, and *forkserver* added for some POSIX platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

在 3.8 版的變更: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725.

在 3.14 版的變更: On POSIX platforms the default start method was changed from *fork* to *forkserver* to retain the performance but avoid common multithreaded process incompatibilities. See gh-84559.

On POSIX using the *spawn* or *forkserver* start methods will also start a *resource tracker* process which tracks the unlinked named system resources (such as named semaphores or `SharedMemory` objects) created by processes of the program. When all processes have exited the resource tracker unlinks any remaining tracked object. Usually there should be none, but if a process was killed by a signal there may be some "leaked" resources. (Neither leaked semaphores nor shared memory segments will be automatically unlinked until the next reboot. This is problematic for both objects because the system allows only a limited number of named semaphores, and shared memory segments occupy some space in the main memory.)

To select a start method you use the `set_start_method()` in the `if __name__ == '__main__'` clause of the main module. For example:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` should not be used more than once in the program.

Alternatively, you can use `get_context()` to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
```

(繼續下一頁)

(繼續上一頁)

```
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the *fork* context cannot be passed to processes started using the *spawn* or *forkserver* start methods.

Libraries using *multiprocessing* or *ProcessPoolExecutor* should be designed to allow their users to provide their own multiprocessing context. Using a specific context of your own within a library can lead to incompatibilities with the rest of the library user's application. Always document if your library requires a specific start method.

⚠ 警告

The `'spawn'` and `'forkserver'` start methods generally cannot be used with "frozen" executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on POSIX systems. The `'fork'` start method may work if code does not use threads.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The *Queue* class is a near clone of *queue.Queue*. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # 印出 "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe. Any object put into a *multiprocessing* queue will be serialized.

Pipes

The *Pipe()* function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # 印出 "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by *Pipe()* represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted

if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

The `send()` method serializes the object and `recv()` re-creates the object.

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from *threading*. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then *multiprocessing* provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using *Value* or *Array*. For example, the following code

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The `'d'` and `'i'` arguments used when creating `num` and `arr` are typecodes of the kind used by the *array* module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. These shared

objects will be process and thread-safe.

For more flexibility in using shared memory one can use the *multiprocessing.sharedctypes* module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by *Manager()* controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by *Manager()* will support types *list*, *dict*, *set*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* and *Array*. For example,

```
from multiprocessing import Process, Manager

def f(d, l, s):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()
    s.add('a')
    s.add('b')

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
        s = manager.set()

        p = Process(target=f, args=(d, l, s))
        p.start()
        p.join()

        print(d)
        print(l)
        print(s)
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
{'a', 'b'}
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The *Pool* class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

舉例來🄵:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
```

(繼續下一頁)

(繼續上一頁)

```
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 seconds
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Note that the methods of a pool should only ever be used by the process which created it.

備𝔽

Functionality within this package requires that the `__main__` module be importable by the children. This is covered in *Programming guidelines* however it is worth pointing out here. This means that some examples, such as the *`multiprocessing.pool.Pool`* examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the parent process somehow.)

18.2.2 Reference

The *multiprocessing* package mostly replicates the API of the *threading* module.

Process 與例外

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

Process objects represent activity that is run in a separate process. The *Process* class has equivalents of all the methods of *threading.Thread*.

The constructor should always be called with keyword arguments. *group* should always be None; it exists solely for compatibility with *threading.Thread*. *target* is the callable object to be invoked by the *run()* method. It defaults to None, meaning nothing is called. *name* is the process name (see *name* for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process *daemon* flag to True or False. If None (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*. The *args* argument, which defaults to (), can be used to specify a list or tuple of the arguments to pass to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (super().__init__()) before doing anything else to the process.

備🄵

In general, all arguments to *Process* must be picklable. This is frequently observed when trying to create a *Process* or use a *concurrent.futures.ProcessPoolExecutor* from a REPL with a locally defined *target* function.

Passing a callable object defined in the current REPL session causes the child process to die via an uncaught *AttributeError* exception when starting as *target* must have been defined within an importable module in order to be loaded during unpickling.

Example of this uncatchable error from the child:

```
>>> import multiprocessing as mp
>>> def knigit():
...     print("Ni!")
...
>>> process = mp.Process(target=knigit)
>>> process.start()
>>> Traceback (most recent call last):
  File ".../multiprocessing/spawn.py", line ..., in spawn_main
  File ".../multiprocessing/spawn.py", line ..., in _main
AttributeError: module '__main__' has no attribute 'knigit'
>>> process
<SpawnProcess name='SpawnProcess-1' pid=379473 parent=378707 stopped exitcode=1>
```

See *The spawn and forkserver start methods*. While this restriction is not true if using the "fork" start method, as of Python 3.14 that is no longer the default on any platform. See *Contexts and start methods*. See also gh-132898.

在 3.3 版的變更: 新增 *daemon* 參數。

run()

Method representing the process's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using a list or tuple as the *args* argument passed to *Process* achieves the same effect.

範例:

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

Start the process's activity.

This must be called at most once per process object. It arranges for the object's *run()* method to be invoked in a separate process.

join([*timeout*])

If the optional argument *timeout* is `None` (the default), the method blocks until the process whose *join()* method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds. Note that the method returns `None` if its process terminates or if the method times out. Check the process's *exitcode* to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form 'Process-N_1:N_2:...:N_k' is constructed, where each N_k is the N-th child of its parent.

is_alive()

Return whether the process is alive.

Roughly, a process object is alive from the moment the *start()* method returns until the child process terminates.

daemon

The process's daemon flag, a Boolean value. This must be set before *start()* is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the *threading.Thread* API, *Process* objects also support the following attributes and methods:

pid

Return the process ID. Before the process is spawned, this will be `None`.

exitcode

The child's exit code. This will be `None` if the process has not yet terminated.

If the child's *run()* method returned normally, the exit code will be 0. If it terminated via *sys.exit()* with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within *run()*, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value *-N*.

authkey

The process's authentication key (a byte string).

When *multiprocessing* is initialized the main process is assigned a random string using *os.urandom()*.

When a *Process* object is created, it will inherit the authentication key of its parent process, although this may be changed by setting *authkey* to another byte string.

參🄵*Authentication keys*。

sentinel

A numeric handle of a system object which will become "ready" when the process ends.

You can use this value if you want to wait on several events at once using *multiprocessing.connection.wait()*. Otherwise calling *join()* is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On POSIX, this is a file descriptor usable with primitives from the *select* module.

在 3.3 版被加入.

interrupt()

Terminate the process. Works on POSIX using the *SIGINT* signal. Behavior on Windows is undefined.

By default, this terminates the child process by raising *KeyboardInterrupt*. This behavior can be altered by setting the respective signal handler in the child process *signal.signal()* for *SIGINT*.

Note: if the child process catches and discards *KeyboardInterrupt*, the process will not be terminated.

Note: the default behavior will also set *exitcode* to `1` as if an uncaught exception was raised in the child process. To have a different *exitcode* you may simply catch *KeyboardInterrupt* and call `exit(your_code)`.

在 3.14 版被加入.

terminate()

Terminate the process. On POSIX this is done using the *SIGTERM* signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated -- they will simply become orphaned.

> **⚠ 警告**
>
> If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

kill()

Same as *terminate()* but using the `SIGKILL` signal on POSIX.

在 3.7 版被加入.

close()

Close the *Process* object, releasing all resources associated with it. *ValueError* is raised if the underlying process is still running. Once *close()* returns successfully, most other methods and attributes of the *Process* object will raise *ValueError*.

在 3.7 版被加入.

Note that the *start()*, *join()*, *is_alive()*, *terminate()* and *exitcode* methods should only be called by the process that created the process object.

Example usage of some of the methods of *Process*:

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

The base class of all *multiprocessing* exceptions.

exception multiprocessing.**BufferTooShort**

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of *BufferTooShort* then `e.args[0]` will give the message as a byte string.

exception multiprocessing.**AuthenticationError**

Raised when there is an authentication error.

exception multiprocessing.**TimeoutError**

Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use *Pipe()* (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The *Queue*, *SimpleQueue* and *JoinableQueue* types are multi-producer, multi-consumer FIFO queues modelled on the *queue.Queue* class in the standard library. They differ in that *Queue* lacks the *task_done()* and *join()* methods introduced into Python 2.5's *queue.Queue* class.

If you use *JoinableQueue* then you **must** call *JoinableQueue.task_done()* for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

One difference from other Python queue implementations, is that *multiprocessing* queues serializes all objects that are put into them using *pickle*. The object return by the get method is a re-created object that does not share memory with the original object.

Note that one can also create a shared queue by using a manager object -- see *Managers*.

備🄵

multiprocessing uses the usual *queue.Empty* and *queue.Full* exceptions to signal a timeout. They are not available in the *multiprocessing* namespace so you need to import them from *queue*.

備🄵

When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties -- if they really bother you then you can instead use a queue created with a *manager*.

(1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's *empty()* method returns *False* and *get_nowait()* can return without raising *queue.Empty*.

(2) If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

警告

If a process is killed using *Process.terminate()* or *os.kill()* while it is trying to use a *Queue*, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

警告

As mentioned above, if a child process has put items on a queue (and it has not used *JoinableQueue.cancel_join_thread*), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See *Programming guidelines*.

For an example of the usage of queues for interprocess communication see 範例.

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`

Returns a pair `(conn1, conn2)` of *Connection* objects representing the ends of a pipe.

If *duplex* is `True` (the default) then the pipe is bidirectional. If *duplex* is `False` then the pipe is unidirectional: `conn1` can only be used for receiving messages and `conn2` can only be used for sending messages.

The `send()` method serializes the object using *pickle* and the `recv()` re-creates the object.

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual *queue.Empty* and *queue.Full* exceptions from the standard library's *queue* module are raised to signal timeouts.

Queue implements all the methods of *queue.Queue* except for *task_done()* and *join()*.

qsize()

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise *NotImplementedError* on platforms like macOS where `sem_getvalue()` is not implemented.

empty()

Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

May raise an *OSError* on closed queues. (not guaranteed)

full()

Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(*obj*[, *block*[, *timeout*]])

Put obj into the queue. If the optional argument *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Full* exception if no free slot was available within that time. Otherwise (*block* is `False`), put an item on the queue if a free slot is immediately available, else raise the *queue.Full* exception (*timeout* is ignored in that case).

在 3.8 版的變更: If the queue is closed, *ValueError* is raised instead of *AssertionError*.

put_nowait(*obj*)

Equivalent to `put(obj, False)`.

get([*block*[, *timeout*]])

Remove and return an item from the queue. If optional args *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Empty* exception if no item was available within that time. Otherwise (block is `False`), return an item if one is immediately available, else raise the *queue.Empty* exception (*timeout* is ignored in that case).

在 3.8 版的變更: If the queue is closed, *ValueError* is raised instead of *OSError*.

get_nowait()

Equivalent to `get(False)`.

multiprocessing.Queue has a few additional methods not found in *queue.Queue*. These methods are usually unnecessary for most code:

close()

Close the queue: release internal resources.

A queue must not be used anymore after it is closed. For example, *get()*, *put()* and *empty()* methods must no longer be called.

The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

Join the background thread. This can only be used after *close()* has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call *cancel_join_thread()* to make *join_thread()* do nothing.

cancel_join_thread()

Prevent *join_thread()* from blocking. In particular, this prevents the background thread from being joined automatically when the process exits -- see *join_thread()*.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

備🄵

This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a *Queue* will result in an *ImportError*. See bpo-3770 for additional information. The same holds true for any of the specialized queue types listed below.

class multiprocessing.**SimpleQueue**

It is a simplified *Queue* type, very close to a locked *Pipe*.

close()

Close the queue: release internal resources.

A queue must not be used anymore after it is closed. For example, *get()*, *put()* and *empty()* methods must no longer be called.

在 3.9 版被加入.

empty()

Return `True` if the queue is empty, `False` otherwise.

Always raises an *OSError* if the SimpleQueue is closed.

get()

Remove and return an item from the queue.

put(*item*)

Put *item* into the queue.

class multiprocessing.**JoinableQueue**([*maxsize*])

JoinableQueue, a *Queue* subclass, is a queue which additionally has *task_done()* and *join()* methods.

task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

join()

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Miscellaneous

multiprocessing.**active_children**()

Return list of all live children of the current process.

Calling this has the side effect of "joining" any processes which have already finished.

multiprocessing.**cpu_count**()

Return the number of CPUs in the system.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with *os.process_cpu_count()* (or `len(os.sched_getaffinity(0))`).

When the number of CPUs cannot be determined a *NotImplementedError* is raised.

> **也參考**
>
> *os.cpu_count()* *os.process_cpu_count()*

在 3.13 版的變更: The return value can also be overridden using the `-X cpu_count` flag or `PYTHON_CPU_COUNT` as this is merely a wrapper around the *os* cpu count APIs.

multiprocessing.**current_process**()

Return the *Process* object corresponding to the current process.

An analogue of *threading.current_thread()*.

multiprocessing.**parent_process**()

Return the *Process* object corresponding to the parent process of the *current_process()*. For the main process, `parent_process` will be `None`.

在 3.8 版被加入.

multiprocessing.**freeze_support**()

Add support for when a program which uses *multiprocessing* has been frozen to produce an executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise *RuntimeError*.

Calling `freeze_support()` has no effect when the start method is not *spawn*. In addition, if the module is being run normally by the Python interpreter (the program has not been frozen), then `freeze_support()` has no effect.

multiprocessing.**get_all_start_methods**()

Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. Not all platforms support all methods. See *Contexts and start methods*.

在 3.4 版被加入.

multiprocessing.**get_context**(*method=None*)

Return a context object which has the same attributes as the *multiprocessing* module.

If *method* is `None` then the default context is returned. Note that if the global start method has not been set, this will set it to the default method. Otherwise *method* should be `'fork'`, `'spawn'`, `'forkserver'`. *ValueError* is raised if the specified start method is not available. See *Contexts and start methods*.

在 3.4 版被加入.

multiprocessing.**get_start_method**(*allow_none=False*)

Return the name of start method used for starting processes.

If the global start method has not been set and *allow_none* is `False`, then the start method is set to the default and the name is returned. If the start method has not been set and *allow_none* is `True` then `None` is returned.

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. See *Contexts and start methods*.

在 3.4 版被加入.

在 3.8 版的變更: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess. See bpo-33725.

multiprocessing.**set_executable**(*executable*)

Set the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

在 3.4 版的變更: Now supported on POSIX when the `'spawn'` start method is used.

在 3.11 版的變更: 接受一個類路徑物件。

multiprocessing.**set_forkserver_preload**(*module_names*)

Set a list of module names for the forkserver main process to attempt to import so that their already imported state is inherited by forked processes. Any *ImportError* when doing so is silently ignored. This can be used as a performance enhancement to avoid repeated work in every process.

For this to work, it must be called before the forkserver process has been launched (before creating a `Pool` or starting a *Process*).

Only meaningful when using the `'forkserver'` start method. See *Contexts and start methods*.

在 3.4 版被加入.

multiprocessing.**set_start_method**(*method*, *force=False*)

Set the method which should be used to start child processes. The *method* argument can be `'fork'`, `'spawn'` or `'forkserver'`. Raises *RuntimeError* if the start method has already been set and *force* is not `True`. If *method* is `None` and *force* is `True` then the start method is set to `None`. If *method* is `None` and *force* is `False` then the context is set to the default context.

Note that this should be called at most once, and it should be protected inside the `if __name__ == '__main__'` clause of the main module.

參🄵*Contexts and start methods*。

在 3.4 版被加入.

備🄵

multiprocessing contains no analogues of *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer*, or *threading.local*.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using *Pipe* -- see also *Listeners and Clients*.

class multiprocessing.connection.**Connection**

send(*obj*)

Send an object to the other end of the connection which should be read using *recv()*.

The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception.

recv()

Return an object sent from the other end of the connection using *send()*. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

fileno()

Return the file descriptor or handle used by the connection.

close()

Close the connection.

This is called automatically when the connection is garbage collected.

poll([*timeout*])

Return whether there is any data available to be read.

If *timeout* is not specified then it will return immediately. If *timeout* is a number then this specifies the maximum time in seconds to block. If *timeout* is `None` then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using *multiprocessing.connection.wait()*.

send_bytes(*buffer*[, *offset*[, *size*]])

Send byte data from a *bytes-like object* as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes([*maxlength*])

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end has closed.

If *maxlength* is specified and the message is longer than *maxlength* then *OSError* is raised and the connection will no longer be readable.

在 3.3 版的變更: This function used to raise *IOError*, which is now an alias of *OSError*.

recv_bytes_into(*buffer*[, *offset*])

Read into *buffer* a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

在 3.3 版的變更: Connection objects themselves can now be transferred between processes using *Connection.send()* and *Connection.recv()*.

Connection objects also now support the context management protocol -- see 情境管理器型🄵. *__enter__()* returns the connection object, and *__exit__()* calls *close()*.

舉例來🄵:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告

The *Connection.recv()* method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using `Pipe()` you should only use the *recv()* and *send()* methods after performing some sort of authentication. See *Authentication keys*.

警告

If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for *threading* module.

Note that one can also create synchronization primitives by using a manager object -- see *Managers*.

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])

A barrier object: a clone of *threading.Barrier*.

在 3.3 版被加入.

class multiprocessing.**BoundedSemaphore**([*value*])

A bounded semaphore object: a close analog of *threading.BoundedSemaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with *Lock.acquire()*.

備🄵

On macOS, this is indistinguishable from *Semaphore* because `sem_getvalue()` is not implemented on that platform.

class multiprocessing.**Condition**([*lock*])

A condition variable: an alias for *threading.Condition*.

If *lock* is specified then it should be a *Lock* or *RLock* object from *multiprocessing*.

在 3.3 版的變更: The *wait_for()* method was added.

class multiprocessing.**Event**

A clone of *threading.Event*.

class multiprocessing.**Lock**

A non-recursive lock object: a close analog of *threading.Lock*. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of *threading.Lock* as it applies to threads are replicated here in *multiprocessing.Lock* as it applies to either processes or threads, except as noted.

Note that *Lock* is actually a factory function which returns an instance of multiprocessing.synchronize.Lock initialized with a default context.

Lock supports the *context manager* protocol and thus may be used in with statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

With the *block* argument set to True (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return True. Note that the name of this first argument differs from that in *threading.Lock.acquire()*.

With the *block* argument set to False, the method call does not block. If the lock is currently in a locked state, return False; otherwise set the lock to a locked state and return True.

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of None (the default) set the timeout period to infinite. Note that the treatment of negative or None values for *timeout* differs from the implemented behavior in *threading.Lock.acquire()*. The *timeout* argument has no practical implications if the *block* argument is set to False and is thus ignored. Returns True if the lock has been acquired or False if the timeout period has elapsed.

release()

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in *threading.Lock.release()* except that when invoked on an unlocked lock, a *ValueError* is raised.

locked()

Return a boolean indicating whether this object is locked right now.

在 3.14 版被加入.

class multiprocessing.**RLock**

A recursive lock object: a close analog of *threading.RLock*. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that *RLock* is actually a factory function which returns an instance of multiprocessing.synchronize.RLock initialized with a default context.

RLock supports the *context manager* protocol and thus may be used in with statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

When invoked with the *block* argument set to True, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of True. Note that there

are several differences in this first argument's behavior compared to the implementation of *threading.RLock.acquire()*, starting with the name of the argument itself.

When invoked with the *block* argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the *timeout* argument are the same as in *Lock.acquire()*. Note that some of these behaviors of *timeout* differ from the implemented behaviors in *threading.RLock.acquire()*.

release()

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An *AssertionError* is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in *threading.RLock.release()*.

locked()

Return a boolean indicating whether this object is locked right now.

在 3.14 版被加入.

class multiprocessing.**Semaphore**([*value*])

A semaphore object: a close analog of *threading.Semaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with *Lock.acquire()*.

> **備🄵**
>
> On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

> **備🄵**
>
> Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an *ImportError*. See bpo-3770 for additional information.

共享的 ctypes 物件

It is possible to create shared objects using shared memory which can be inherited by child processes.

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)

Return a *ctypes* object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the *value* attribute of a *Value*.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *array* module. **args* is passed on to the constructor for the type.

If *lock* is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False`

then access to the returned object will not be automatically protected by a lock, so it will not necessarily be ”process-safe”.

Operations like += which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that *lock* is a keyword-only argument.

`multiprocessing.`**`Array`**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *array* module. If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be ”process-safe”.

Note that *lock* is a keyword only argument.

Note that an array of *ctypes.c_char* has *value* and *raw* attributes which allow one to use it to store and retrieve strings.

`multiprocessing.sharedctypes` 模組

The *multiprocessing.sharedctypes* module provides functions for allocating *ctypes* objects from shared memory which can be inherited by child processes.

> **備🄵**
>
> Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

`multiprocessing.sharedctypes.`**`RawArray`**(*typecode_or_type*, *size_or_initializer*)

Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *array* module. If *size_or_initializer* is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic -- use *Array()* instead to make sure that access is automatically synchronized using a lock.

`multiprocessing.sharedctypes.`**`RawValue`**(*typecode_or_type*, **args*)

Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *array* module. **args* is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic -- use *Value()* instead to make sure that access is automatically synchronized using a lock.

Note that an array of *ctypes.c_char* has `value` and `raw` attributes which allow one to use it to store and retrieve strings -- see documentation for *ctypes*.

`multiprocessing.sharedctypes.`**`Array`**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

The same as *RawArray()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

`multiprocessing.sharedctypes.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

The same as *RawValue()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

`multiprocessing.sharedctypes.`**`copy`**(*obj*)

Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

`multiprocessing.sharedctypes.`**`synchronized`**(*obj*[, *lock*])

Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is `None` (the default) then a *multiprocessing.RLock* object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

在 3.5 版的變更: Synchronized objects support the *context manager* protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of *ctypes.Structure*.)

ctypes	sharedctypes using type	sharedctypes using typecode
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
```

(繼續下一頁)

(繼續上一頁)

```
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

`multiprocessing.`**`Manager`**`()`

Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the *multiprocessing.managers* module:

class multiprocessing.managers.**BaseManager**(*address=None*, *authkey=None*, *serializer='pickle'*, *ctx=None*, ***, *shutdown_timeout=1.0*)

建立一個 BaseManager 物件。

Once created one should call *start()* or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

address is the address on which the manager process listens for new connections. If *address* is `None` then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is `None` then `current_process().authkey` is used. Otherwise *authkey* is used and it must be a byte string.

serializer must be `'pickle'` (use *pickle* serialization) or `'xmlrpclib'` (use *xmlrpc.client* serialization).

ctx is a context object, or `None` (use the current context). See the `get_context()` function.

shutdown_timeout is a timeout in seconds used to wait until the process used by the manager completes in the *shutdown()* method. If the shutdown times out, the process is terminated. If terminating the process also times out, the process is killed.

在 3.11 版的變更: 新增 *shutdown_timeout* 參數。

start(*[initializer[, initargs]]*)

Start a subprocess to start the manager. If *initializer* is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

get_server()

Returns a `Server` object which represents the actual server under the control of the Manager. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an *address* attribute.

connect()

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()

Stop the process used by the manager. This is only available if *start()* has been used to start the server process.

This can be called multiple times.

register(*typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]]*)

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a "type identifier" which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the *connect()* method, or if the *create_method* argument is `False` then this can be left as `None`.

proxytype is a subclass of *BaseProxy* which is used to create proxies for shared objects with this *typeid*. If `None` then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using *BaseProxy._callmethod()*. (If *exposed* is `None` then `proxytype._exposed_` is used instead if it exists.) In the case where no exposed list is specified, all "public methods" of the shared object will be accessible. (Here a "public method" means any attribute which has a `__call__()` method and whose name does not begin with `'_'`.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If *method_to_typeid* is `None` then `proxytype._method_to_typeid_` is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

BaseManager instances also have one read-only property:

address

The address used by the manager.

在 3.3 版的變更: Manager objects support the context management protocol -- see 情境管理器型🄵. *__enter__()* starts the server process (if it has not already started) and then returns the manager object. *__exit__()* calls *shutdown()*.

In previous versions *__enter__()* did not start the manager's server process if it was not already started.

class multiprocessing.managers.**SyncManager**

A subclass of *BaseManager* which can be used for the synchronization of processes. Objects of this type are returned by *multiprocessing.Manager()*.

Its methods create and return *Proxy Objects* for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])

Create a shared *threading.Barrier* object and return a proxy for it.

在 3.3 版被加入.

BoundedSemaphore([*value*])

Create a shared *threading.BoundedSemaphore* object and return a proxy for it.

Condition([*lock*])

Create a shared *threading.Condition* object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a *threading.Lock* or *threading.RLock* object.

在 3.3 版的變更: The *wait_for()* method was added.

Event()

Create a shared *threading.Event* object and return a proxy for it.

Lock()

Create a shared *threading.Lock* object and return a proxy for it.

Namespace()

Create a shared *Namespace* object and return a proxy for it.

Queue([*maxsize*])

Create a shared *queue.Queue* object and return a proxy for it.

RLock()

Create a shared *threading.RLock* object and return a proxy for it.

Semaphore([*value*])

Create a shared *threading.Semaphore* object and return a proxy for it.

Array(*typecode*, *sequence*)

Create an array and return a proxy for it.

Value(*typecode*, *value*)

Create an object with a writable `value` attribute and return a proxy for it.

dict()

dict(*mapping*)

dict(*sequence*)

Create a shared *dict* object and return a proxy for it.

list()

list(*sequence*)

Create a shared *list* object and return a proxy for it.

set()

set(*sequence*)

set(*mapping*)

Create a shared *set* object and return a proxy for it.

在 3.14 版被加入: *set* support was added.

在 3.6 版的變更: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the *SyncManager*.

class multiprocessing.managers.**Namespace**

A type that can register with *SyncManager*.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of *BaseManager* and uses the *register()* classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # 印出 7
        print(maths.mul(7, 8))         # 印出 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying *str()* to a proxy will return the representation of the referent, whereas applying *repr()* will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain *Proxy Objects*. This permits nesting of these managed lists, dicts, and other *Proxy Objects*:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) *list* or *dict* objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested *Proxy Objects* for most use cases but also demonstrates a level of control over the synchronization.

備🄵

The proxy types in *multiprocessing* do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

class multiprocessing.managers.**BaseProxy**

Proxy objects are instances of subclasses of *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Call and return the result of a method of the proxy's referent.

If `proxy` is a proxy whose referent is `obj` then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object -- see documentation for the *method_to_typeid* argument of *BaseManager.register()*.

If an exception is raised by the call, then is re-raised by *_callmethod()*. If some other exception is raised in the manager's process then this is converted into a `RemoteError` exception and is raised by *_callmethod()*.

Note in particular that an exception will be raised if *methodname* has not been *exposed*.

An example of the usage of *_callmethod()*:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # 等價於 l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # 等價於 l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

__repr__()

Return a representation of the proxy object.

__str__()

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the *Pool* class.

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by *os.process_cpu_count()* is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()` or the *Pool()* method of a context object. In both cases *context* is set appropriately.

Note that the methods of the pool object should only be called by the process which created the pool.

警告

multiprocessing.pool objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling *close()* and *terminate()* manually. Failure to do this can lead to the process hanging on finalization.

Note that it is **not correct** to rely on the garbage collector to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

在 3.2 版的變更: 新增 *maxtasksperchild* 參數。

在 3.4 版的變更: 新增 *context* 參數。

在 3.13 版的變更: *processes* uses *os.process_cpu_count()* by default, instead of *os.cpu_count()*.

備🄵

Worker processes within a *Pool* typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the *Pool* exposes this ability to the end user.

apply(*func*[, *args*[, *kwds*]])

Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, *apply_async()* is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])

A variant of the *apply()* method which returns a *AsyncResult* object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

map(*func*, *iterable*[, *chunksize*])

A parallel equivalent of the *map()* built-in function (it supports only one *iterable* argument though, for multiple iterables see *starmap()*). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

Note that it may cause high memory usage for very long iterables. Consider using *imap()* or *imap_unordered()* with explicit *chunksize* option for better efficiency.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A variant of the *map()* method which returns a *AsyncResult* object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])

A lazier version of `map()`.

The *chunksize* argument is the same as the one used by the `map()` method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of `1`.

Also if *chunksize* is `1` then the `next()` method of the iterator returned by the `imap()` method has an optional *timeout* parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

imap_unordered(*func*, *iterable*[, *chunksize*])

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".)

starmap(*func*, *iterable*[, *chunksize*])

Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of `[(1,2), (3, 4)]` results in `[func(1,2), func(3,4)]`.

在 3.3 版被加入.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A combination of `starmap()` and `map_async()` that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

在 3.3 版被加入.

close()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

join()

Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

在 3.3 版的變更: Pool objects now support the context management protocol -- see 情境管理器型🄵. `__enter__()` returns the pool object, and `__exit__()` calls `terminate()`.

class multiprocessing.pool.**AsyncResult**

The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

get([*timeout*])

Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

wait([*timeout*])

Wait until the result is available or until *timeout* seconds pass.

ready()

Return whether the call has completed.

successful()

Return whether the call completed without raising an exception. Will raise *ValueError* if the result is not ready.

在 3.7 版的變更: If the result is not ready, *ValueError* is raised instead of *AssertionError*.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a single␣
↪process
        print(result.get(timeout=1))         # prints "100" unless your computer is *very*␣
↪slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is *very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

However, the *multiprocessing.connection* module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the *hmac* module, and for polling multiple connections at the same time.

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise *AuthenticationError* is raised.

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)

Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then *AuthenticationError* is raised.

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])

Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See *Address Formats*)

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])

A wrapper for a bound socket or Windows named pipe which is 'listening' for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

> **備🄵**
>
> If an address of '0.0.0.0' is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use '127.0.0.1'.

family is the type of socket (or named pipe) to use. This can be one of the strings `'AF_INET'` (for a TCP socket), `'AF_UNIX'` (for a Unix domain socket) or `'AF_PIPE'` (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is `None` then the family is inferred from the format of *address*. If *address* is also `None` then a default is chosen. This default is the family which is assumed to be the fastest available. See *Address Formats*. Note that if *family* is `'AF_UNIX'` and address is `None` then the socket will be created in a private temporary directory created using *tempfile.mkstemp()*.

If the listener object uses a socket then *backlog* (1 by default) is passed to the *listen()* method of the socket once it has been bound.

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a *Connection* object. If authentication is attempted and fails, then *AuthenticationError* is raised.

close()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address

The address which is being used by the Listener object.

last_accepted

The address from which the last accepted connection came. If this is unavailable then it is `None`.

在 3.3 版的變更: Listener objects now support the context management protocol -- see 情境管理器型🄵. *__enter__()* returns the listener object, and *__exit__()* calls *close()*.

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)

Wait till an object in *object_list* is ready. Returns the list of those objects in *object_list* which are ready. If *timeout* is a float then the call blocks for at most that many seconds. If *timeout* is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both POSIX and Windows, an object can appear in *object_list* if it is

- a readable *Connection* object;
- a connected and readable *socket.socket* object; or
- the *sentinel* attribute of a *Process* object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

POSIX: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if *select.select()* is interrupted by a signal, it can raise *OSError* with an error number of `EINTR`, whereas *wait()* will not.

Windows: An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a *fileno()* method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

在 3.3 版被加入.

Examples

The following server code creates a listener which uses `'secret password'` as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

The following code uses *wait()* to wait for messages from multiple processes at once:

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
```

(繼續下一頁)

(繼續上一頁)

```
            msg = r.recv()
        except EOFError:
            readers.remove(r)
        else:
            print(msg)
```

Address Formats

- An `'AF_INET'` address is a tuple of the form `(hostname, port)` where *hostname* is a string and *port* is an integer.
- An `'AF_UNIX'` address is a string representing a filename on the filesystem.
- An `'AF_PIPE'` address is a string of the form `r'\\.\pipe\`*PipeName*`'`. To use *`Client()`* to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\`*`ServerName`*`\pipe\`*`PipeName`*`'` instead.

Note that any string beginning with two backslashes is assumed by default to be an `'AF_PIPE'` address rather than an `'AF_UNIX'` address.

Authentication keys

When one uses *`Connection.recv`*, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore *`Listener`* and *`Client()`* use the *`hmac`* module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of `current_process().authkey` is used (see *`Process`*). This value will be automatically inherited by any *`Process`* object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using *`os.urandom()`*.

Logging

Some support for logging is available. Note, however, that the *`logging`* package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

`multiprocessing.`**`get_logger`**`()`

Returns the logger used by *`multiprocessing`*. If necessary, a new one will be created.

When first created the logger has level *`logging.NOTSET`* and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger -- any other customization of the logger will not be inherited.

`multiprocessing.`**`log_to_stderr`**`(`*level=None*`)`

This function performs a call to *`get_logger()`* but in addition to returning the logger created by get_logger, it adds a handler which sends output to *`sys.stderr`* using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
```

(繼續下一頁)

(繼續上一頁)

```
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

For a full table of logging levels, see the *logging* module.

multiprocessing.dummy 模組

multiprocessing.dummy replicates the API of *multiprocessing* but is no more than a wrapper around the *threading* module.

In particular, the `Pool` function provided by *multiprocessing.dummy* returns an instance of *ThreadPool*, which is a subclass of *Pool* that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

A thread pool object which controls a pool of worker threads to which jobs can be submitted. *ThreadPool* instances are fully interface compatible with *Pool* instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling *close()* and *terminate()* manually.

processes is the number of worker threads to use. If *processes* is `None` then the number returned by *os.process_cpu_count()* is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

Unlike *Pool*, *maxtasksperchild* and *context* cannot be provided.

> **備🄵**
>
> A *ThreadPool* shares the same interface as *Pool*, which is designed around a pool of processes and predates the introduction of the *concurrent.futures* module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, *AsyncResult*, that is not understood by any other libraries.
>
> Users should generally prefer to use *concurrent.futures.ThreadPoolExecutor*, which has a simpler interface that was designed around threads from the start, and which returns *concurrent.futures.Future* instances that are compatible with many other libraries, including *asyncio*.

18.2.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using *multiprocessing*.

All start methods

The following applies to all start methods.

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

On POSIX when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or *active_children()* is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's *Process.is_alive* will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

When using the *spawn* or *forkserver* start methods many types from *multiprocessing* need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the *Process.terminate* method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using *Process.terminate* on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the "feeder" thread to the underlying pipe. (The child process can call the *Queue.cancel_join_thread* method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On POSIX using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing *sys.stdin* with a ”file like object”

multiprocessing originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method --- this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace *sys.stdin()* with a ”file-like object” with output buffering. This danger is that if multiple processes call *close()* on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155, bpo-5313 and bpo-5331

The *spawn* and *forkserver* start methods

There are a few extra restrictions which don’t apply to the *fork* start method.

More picklability

Ensure that all arguments to *Process* are picklable. Also, if you subclass `Process.__init__`, you must make sure that instances will be picklable when the *Process.start* method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that *Process.start* was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

For example, using the *spawn* or *forkserver* start method running the following module would fail with a *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the "entry point" of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module's `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

18.2.4 范例

Demonstration of how to create and use customized managers and proxies:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
```

(繼續下一頁)

(繼續上一頁)

```
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)
```

(繼續下一頁)

(繼續上一頁)

```
    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

使用*Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
```

(繼續下一頁)

(繼續上一頁)

```
    #

    TASKS = [(mul, (i, 7)) for i in range(10)] + \
            [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print('Ordered results using pool.apply_async():')
    for r in results:
        print('\t', r.get())
    print()

    print('Ordered results using pool.imap():')
    for x in imap_it:
        print('\t', x)
    print()

    print('Unordered results using pool.imap_unordered():')
    for x in imap_unordered_it:
        print('\t', x)
    print()

    print('Ordered results using pool.map() --- will block till complete:')
    for x in pool.map(calculatestar, TASKS):
        print('\t', x)
    print()

    #
    # Test error handling
    #

    print('Testing error handling:')

    try:
        print(pool.apply(f, (5,)))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.apply()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(pool.map(f, list(range(10))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.map()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(list(pool.imap(f, list(range(10)))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from list(pool.imap())')
    else:
        raise AssertionError('expected ZeroDivisionError')

    it = pool.imap(f, list(range(10)))
    for i in range(10):
        try:
            x = next(it)
        except ZeroDivisionError:
```

(繼續下一頁)

(繼續上一頁)

```
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)
```

(繼續下一頁)

(繼續上一頁)

```
#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == '__main__':
    freeze_support()
    test()
```

18.3 multiprocessing.shared_memory --- 對於共享記憶體的跨行程直接存取

原始碼： Lib/multiprocessing/shared_memory.py

在 3.8 版被加入.

該模組提供了一個*SharedMemory* 類🄵，用於分配和管理被多核心或對稱多處理器 (symmetric multiprocessor, SMP) 機器上的一個或多個行程存取的共享記憶體。🄵了協助共享記憶體的生命🄵期管理，特🄵是跨不同行程的管理，*multiprocessing.managers* 模組中還提供了一個*BaseManager* 子類🄵*SharedMemoryManager*。

在此模組中，共享記憶體是指「POSIX 風格」的共享記憶體區塊（🄵管不一定如此明確實作），而不是指「分散式共享記憶體 (distributed shared memory)」。這種型🄵的共享記憶體允許不同的行程🄵在地讀取和寫入揮發性記憶體 (volatile memory) 的公開（或共享）區域。通常行程只能存取自己的行程記憶體空間，但共享記憶體允許在行程之間共享資料，從而避免需要跨行程傳遞資料的情境。與透過硬碟或 socket 或其他需要序列化/還原序列化 (serialization/deserialization) 和🄵🄵資料的通訊方式以共享資料相比，直接透過記憶體共享資料可以提供顯著的性能優勢。

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*, ***, *track=True*)

建立 SharedMemory 類🄵的實例，用於建立新的共享記憶體區塊或附加到現有的共享記憶體區塊。每個共享記憶體區塊都被分配了一個唯一的名稱。透過這種方式，一個行程可以建立具有特定名稱的共享記憶體區塊，而不同的行程可以使用該相同名稱附加到同一共享記憶體塊。

作🄵跨行程共享資料的資源，共享記憶體區塊的壽命可能比建立它們的原始行程還要長。當一個行程不再需要存取但其他行程可能仍需要的共享記憶體區塊時，應該呼叫*close()* 方法。當任何行程不再需要共享記憶體區塊時，應呼叫*unlink()* 方法以確保正確清理。

參數

- **name** (str | *None*) -- 所請求的共享記憶體的唯一名稱，指定🄵字串。建立新的共享記憶體區塊時，如果名稱提供🄵 None（預設值），則會生成一個新的名稱。
- **create** (bool) -- 控制是否建立新的共享記憶體區塊 (True) 或附加現有的共享記憶體區塊 (False)。
- **size** (int) -- 指定建立新共享記憶體區塊時請求的位元組數。由於某些平台會根據該平台的記憶體頁 (memory page) 大小來選擇分配記憶體區塊，因此共享記憶體區塊的確切大小可能大於或等於請求的大小。當附加到現有共享記憶體區塊時，size 參數將被忽略。
- **track** (bool) -- When True, register the shared memory block with a resource tracker process on platforms where the OS does not do this automatically. The resource tracker ensures proper cleanup of the shared memory even if all other processes with access to the memory exit without doing so. Python processes created from a common ancestor using *multiprocessing* facilities share a single resource tracker process, and the lifetime of shared memory segments is handled automatically among these processes. Python processes created in any other way will receive their own resource tracker when accessing shared memory with *track* enabled. This will cause the shared memory to be deleted by the resource tracker of the first process that terminates. To avoid this issue, users of *subprocess* or standalone Python processes should set *track* to False when there is already another process in place that does the bookkeeping. *track* is ignored on Windows, which has its own tracking and automatically deletes shared memory when all handles to it have been closed.

在 3.13 版的變更: 新增 *track* 參數。

close()

Close the file descriptor/handle to the shared memory from this instance. *close()* should be called once access to the shared memory block from this instance is no longer needed. Depending on operating system, the underlying memory may or may not be freed even if all handles to it have been closed. To ensure proper cleanup, use the *unlink()* method.

unlink()

Delete the underlying shared memory block. This should be called only once per shared memory block regardless of the number of handles to it, even in other processes. *unlink()* and *close()* can be called in any order, but trying to access data inside a shared memory block after *unlink()* may result in memory access errors, depending on platform.

This method has no effect on Windows, where the only way to delete a shared memory block is to close all handles.

buf

共享記憶體區塊🄴容的記憶體視圖 (memoryview)。

name

對共享記憶體區塊之唯一名稱的唯讀存取。

size

對共享記憶體區塊大小（以位元組🄴單位）的唯讀存取。

以下範例示範了*SharedMemory* 實例的低階使用方式：

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple at once
>>> buffer[4] = 100                           # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

以下範例示範了*SharedMemory* 類🄴與 NumPy 陣列的實際用法：從兩個不同的 Python shell 存取相同的 `numpy.ndarray`：

```
>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
```

(繼續下一頁)

(繼續上一頁)

```
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

multiprocessing.managers.BaseManager 的子類🄵，可用於跨行程管理共享記憶體區塊。

在 SharedMemoryManager 實例上呼叫*start()* 會🄵動一個新行程。這個新行程的唯一目的是管理那些透過它建立出的所有共享記憶體區塊的生命🄵期。要觸發釋放該行程管理的所有共享記憶體區塊，請在實例上呼叫*shutdown()*，這會觸發對該行程管理的所有*SharedMemory* 物件的*unlink()* 呼叫，然後再停止這個行程。透過 SharedMemoryManager 建立 SharedMemory 實例，我們無需手動追🄵和觸發共享記憶體資源的釋放。

此類🄵提供了用於建立和回傳*SharedMemory* 實例以及建立由共享記憶體支援的類串列物件 (*ShareableList*) 的方法。

請參🄵*BaseManager* 了解繼承的 *address* 和 *authkey* 可選輸入引數的描述以及如何使用它們從其他行程連接到現有的 SharedMemoryManager 服務。

SharedMemory(*size*)

建立🄵回傳一個新的*SharedMemory* 物件，該物件具有指定的 *size*（以位元組🄵單位）。

ShareableList(*sequence*)

建立🄵回傳一個新的*ShareableList* 物件，該物件由輸入 *sequence* 中的值初始化。

以下範例示範了*SharedMemoryManager* 的基本作用機制：

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
```

(繼續下一頁)

(繼續上一頁)

```
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and another_sl
```

以下範例描述了一種可能更方便的模式，即透過 `with` 陳述式使用*SharedMemoryManager* 物件，以確保所有共享記憶體區塊不再被需要後都被釋放：

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

在 `with` 陳述式中使用*SharedMemoryManager* 時，當 `with` 陳述式的程式碼區塊執行完畢時，使用該管理器建立的共享記憶體區塊都會被釋放。

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

提供一個類似 list 的可變物件，其中儲存的所有值都儲存在共享記憶體區塊中。這將可儲存值限制🄵以下🄵建資料型🄵：

- *int*（有符號 64 位元）
- *float*
- *bool*
- *str*（編碼🄵 UTF-8 時每個小於 10M 位元組）
- *bytes*（每個小於 10M 位元組）
- `None`

它也與🄵建*list* 型🄵明顯不同，因🄵這些 list 不能更改其總長度（即🄵有 `append()`、`insert()` 等）🄵且不支援通過切片動態建立新的 `ShareableList` 實例。

sequence 用於填充 (populate) 一個充滿值的新 `ShareableList`。設定🄵 `None` 以透過其唯一的共享記憶體名稱來附加到已經存在的 `ShareableList`。

如*SharedMemory* 的定義中所述，*name* 是被請求之共享記憶體的唯一名稱。當附加到現有的 `ShareableList` 時，指定其共享記憶體區塊的唯一名稱，同時將 *sequence* 設定🄵 `None`。

> **備🄵**
>
> *bytes* 和*str* 值存在一個已知問題。如果它們以 `\x00` nul 位元組或字元結尾，那🄵當透過索引從 `ShareableList` 中取得它們時，這些位元組或字元可能會被默默地🄵離 *(silently stripped)*。這種 `.rstrip(b'\x00')` 行🄵被認🄵是一個錯誤，將來可能會消失。請參🄵 gh-106939。

對於去除尾隨空值 (rstripping of trailing nulls) 會出問題的應用程式，變通解法 (workaround) 是始終無條件地在儲存時於此類值的末尾追加一個額外非 0 位元組，🄵在取得時也無條件地🄵除它：

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00', b'\x03\x02\x01\x00\x00\x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
```

(繼續下一頁)

(繼續上一頁)

```
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07', b'\x03\x02\x01\x00\x00\x00\x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

回傳 *value* 出現的次數。

index(*value*)

回傳 *value* 的第一個索引位置。如果 *value* 不存在，則引發*ValueError*。

format

唯讀屬性，包含所有目前有儲存的值所使用的*struct* 打包格式。

shm

儲存值的*SharedMemory* 實例。

以下範例示範了*ShareableList* 實例的基本用法：

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True, 42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>, <class
↪'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

以下範例描述了一個、兩個或多個行程如何透過提供後面的共享記憶體區塊名稱來存取同一個*ShareableList*：

```
>>> b = shared_memory.ShareableList(range(5))          # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)   # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
```

(繼續下一頁)

(繼續上一頁)

```
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

以下範例示範了如果需要，可以對*ShareableList*（和底層*SharedMemory*）物件進行 pickle 和 unpickle。請注意，它仍然是相同的共享物件。發生這種情🄴是因🄴反序列化的物件具有相同的唯一名稱，🄵且只是附加到具有相同名稱的現有物件（如果該物件仍然存在）：

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.shm.unlink()
```

18.4 concurrent 套件

此套件包含下列模組：

- *concurrent.futures* -- 🄵動平行任務
- *concurrent.interpreters* -- 在同一行程中使用多個直譯器

18.5 concurrent.futures --- 🄵動平行任務

在 3.2 版被加入.

原　　始　　碼： Lib/concurrent/futures/thread.py、Lib/concurrent/futures/process.py 與 Lib/concurrent/futures/interpreter.py

---

concurrent.futures 模組提供了一個高階介面來非同步地 (asynchronously) 執行可呼叫物件 (callable) 。

非同步執行可以透過*ThreadPoolExecutor* 或*InterpreterPoolExecutor* 來使用執行緒 (thread) 執行，或透過*ProcessPoolExecutor* 來使用單獨行程 (process) 執行。兩者都實作了相同的介面，該介面由抽象的*Executor* 類🄴定義。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄴*WebAssembly 平台*。

18.5.1 Executor 物件

class concurrent.futures.**Executor**

提供非同步執行呼叫方法的抽象類🄵。不應直接使用它，而應透過其具體子類🄵來使用。

submit(*fn*, */*, **args*, ***kwargs*)

🄵可呼叫物件 *fn* 排程來以 `fn(*args, **kwargs)` 的形式執行🄵回傳一個表示可呼叫的執行的*Future* 物件。

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*, *buffersize=None*)

類似於*map(fn, *iterables)*，除了：

- The *iterables* are collected immediately rather than lazily, unless a *buffersize* is specified to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer.
- *fn* 是非同步執行的，🄵且對 *fn* 的多次呼叫可以🄵行處理。

如果*__next__()* 被呼叫，且在原先呼叫*Executor.map()* 的 *timeout* 秒後結果仍不可用，回傳的🄵代器就會引發*TimeoutError*。*timeout* 可以是整數或浮點數。如果未指定 *timeout* 或🄵 `None`，則等待時間就不會有限制。

如果 *fn* 呼叫引發例外，則當從🄵代器中檢索到它的值時將引發該例外。

使用*ProcessPoolExecutor* 時，此方法將 *iterables* 分成許多分塊 (chunks)，🄵將其作🄵獨立的任務來提交給池 (pool)。可以透過將 *chunksize* 設定🄵正整數來指定這些分塊的（約略）大小。對於非常長的可🄵代物件，*chunksize* 使用較大的值（與預設大小 1 相比）可以顯著提高性能。對於*ThreadPoolExecutor* 和*InterpreterPoolExecutor*，*chunksize* 無效。

在 3.5 版的變更: 新增 *chunksize* 參數。

在 3.14 版的變更: 新增 *buffersize* 參數。

shutdown(*wait=True*, ***, *cancel_futures=False*)

向 executor 發出訊號 (signal)，表明它應該在目前未定 (pending) 的 future 完成執行時釋放它正在使用的任何資源。在關閉後呼叫*Executor.submit()* 和*Executor.map()* 將引發*RuntimeError*。

如果 *wait* 🄵 `True` 則此方法將不會回傳，直到所有未定的 futures 完成執行🄵且與 executor 關聯的資源都被釋放。如果 *wait* 🄵 `False` 則此方法將立即回傳，🄵且當所有未定的 future 執行完畢時，與 executor 關聯的資源將被釋放。不管 *wait* 的值如何，整個 Python 程式都不會退出，直到所有未定的 futures 執行完畢。

如果 *cancel_futures* 🄵 `True`，此方法將取消 executor 尚未開始運行的所有未定 future。無論 *cancel_futures* 的值如何，任何已完成或正在運行的 future 都不會被取消。

如果 *cancel_futures* 和 *wait* 都🄵 `True`，則 executor 已開始運行的所有 future 將在此方法回傳之前完成。剩余的 future 被取消。

如果使用 `with` 陳述式來將 executor 用作*context manager*，那你就可以不用明確地呼叫此方法，這將會自己關閉*Executor*（如同呼叫*Executor.shutdown()* 時 *wait* 被設定🄵 `True` 般等待）：

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

在 3.9 版的變更: 新增 *cancel_futures*。

18.5.2 ThreadPoolExecutor

ThreadPoolExecutor 是一個*Executor* 子類🄴，它使用執行緒池來非同步地執行呼叫。

當與*Future* 關聯的可呼叫物件等待另一個*Future* 的結果時，可能會發生死鎖 (deadlock)。例如：

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b 永遠不會完成，因🄵它正在等待 a。
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a 永遠不會完成，因🄵它正在等待 b。
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

和：

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # 這將永遠不會完成，因🄵只有一個工作執行緒且
    # 它正在執行這個函式。
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

一個*Executor* 子類🄴，它使用最多有 *max_workers* 個執行緒的池來非同步地執行呼叫。

所有排隊到 ThreadPoolExecutor 的執行緒都將在直譯器退出之前加入。請注意，執行此操作的退出處理程式會在任何使用 atexit 新增的退出處理程式之前執行。這意味著必須捕獲🄴處理主執行緒中的例外，以便向執行緒發出訊號來正常退出 (gracefully exit)。因此，建議不要將 ThreadPoolExecutor 用於長時間運行的任務。

initializer 是一個可選的可呼叫物件，在每個工作執行緒開始時呼叫；*initargs* 是傳遞給 initializer 的引數元組 (tuple)。如果 *initializer* 引發例外，所有目前未定的作業以及任何向池中提交 (submit) 更多作業的嘗試都將引發*BrokenThreadPool*。

在 3.5 版的變更: 如果 *max_workers* 🄵 None 或未給定，它將預設🄵機器上的處理器數量乘以 5，這假定了*ThreadPoolExecutor* 通常用於 I/O 重🄵而非 CPU 密集的作業，🄵且 worker 的數量應該高於*ProcessPoolExecutor* 的 worker 數量。

在 3.6 版的變更: 新增 *thread_name_prefix* 參數以允許使用者控制由池所建立的工作執行緒 (worker thread) 的*threading.Thread* 名稱，以便於除錯。

在 3.7 版的變更: 新增 *initializer* 與 *initargs* 引數。

在 3.8 版的變更: *max_workers* 的預設值改🄵 min(32, os.cpu_count() + 4)。此預設值🄵 I/O 密集任務至少保留了 5 個 worker。它最多使用 32 個 CPU 核心來執行 CPU 密集任務，以釋放 GIL。🄵且它避免了在多核機器上隱晦地使用非常大量的資源。

ThreadPoolExecutor 現在在🄵動 *max_workers* 工作執行緒之前會重用 (reuse) 空🄵的工作執行緒。

在 3.13 版的變更: *max_workers* 的預設值被改🄵 min(32, (os.process_cpu_count() or 1) + 4)。

ThreadPoolExecutor 范例

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistent-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

18.5.3 InterpreterPoolExecutor

The *InterpreterPoolExecutor* class uses a pool of interpreters to execute calls asynchronously. It is a *ThreadPoolExecutor* subclass, which means each worker is running in its own thread. The difference here is that each worker has its own interpreter, and runs each task using that interpreter.

The biggest benefit to using interpreters instead of only threads is true multi-core parallelism. Each interpreter has its own *Global Interpreter Lock*, so code running in one interpreter can run on one CPU core, while code in another interpreter runs unblocked on a different core.

The tradeoff is that writing concurrent code for use with multiple interpreters can take extra effort. However, this is because it forces you to be deliberate about how and when interpreters interact, and to be explicit about what data is shared between interpreters. This results in several benefits that help balance the extra effort, including true multi-core parallelism, For example, code written this way can make it easier to reason about concurrency. Another major benefit is that you don't have to deal with several of the big pain points of using threads, like race conditions.

Each worker's interpreter is isolated from all the other interpreters. "Isolated" means each interpreter has its own runtime state and operates completely independently. For example, if you redirect *sys.stdout* in one interpreter, it will not be automatically redirected to any other interpreter. If you import a module in one interpreter, it is not automatically imported in any other. You would need to import the module separately in interpreter where you need it. In fact, each module imported in an interpreter is a completely separate object from the same module in a different interpreter, including *sys*, *builtins*, and even `__main__`.

Isolation means a mutable object, or other data, cannot be used by more than one interpreter at the same time. That effectively means interpreters cannot actually share such objects or data. Instead, each interpreter must have its own copy, and you will have to synchronize any changes between the copies manually. Immutable objects and data, like the builtin singletons, strings, and tuples of immutable objects, don't have these limitations.

Communicating and synchronizing between interpreters is most effectively done using dedicated tools, like those proposed in **PEP 734**. One less efficient alternative is to serialize with *pickle* and then send the bytes over a shared *socket* or *pipe*.

class concurrent.futures.**InterpreterPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

A *ThreadPoolExecutor* subclass that executes calls asynchronously using a pool of at most *max_workers* threads. Each thread runs tasks in its own interpreter. The worker interpreters are isolated from each other, which means each has its own runtime state and that they can't share any mutable objects or other data. Each interpreter has its own *Global Interpreter Lock*, which means code run with this executor has true multi-core parallelism.

The optional *initializer* and *initargs* arguments have the same meaning as for `ThreadPoolExecutor`: the initializer is run when each worker is created, though in this case it is run in the worker's interpreter. The executor serializes the *initializer* and *initargs* using *pickle* when sending them to the worker's interpreter.

> **備🄵**
>
> The executor may replace uncaught exceptions from *initializer* with *ExecutionFailed*.

Other caveats from parent *ThreadPoolExecutor* apply here.

submit() and *map()* work like normal, except the worker serializes the callable and arguments using *pickle* when sending them to its interpreter. The worker likewise serializes the return value when sending it back.

When a worker's current task raises an uncaught exception, the worker always tries to preserve the exception as-is. If that is successful then it also sets the `__cause__` to a corresponding *ExecutionFailed* instance, which contains a summary of the original exception. In the uncommon case that the worker is not able to preserve the original as-is then it directly preserves the corresponding *ExecutionFailed* instance instead.

18.5.4 ProcessPoolExecutor

ProcessPoolExecutor 類🄵是一個*Executor* 的子類🄵，它使用行程池來非同步地執行呼叫。*ProcessPoolExecutor* 使用了*multiprocessing* 模組，這允許它避開全域直譯器鎖 *(Global Interpreter Lock)*，但也意味著只能執行和回傳可被 pickle 的 (picklable) 物件。

`__main__` 模組必須可以被工作子行程 (worker subprocess) 引入。這意味著*ProcessPoolExecutor* 將無法在交互式直譯器 (interactive interpreter) 中工作。

從提交給*ProcessPoolExecutor* 的可呼叫物件中呼叫*Executor* 或*Future* 方法將導致死鎖。

Note that the restrictions on functions and arguments needing to picklable as per *multiprocessing.Process* apply when using *submit()* and *map()* on a *ProcessPoolExecutor*. A function defined in a REPL or a lambda should not be expected to work.

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

一個*Executor* 子類🄵，它使用了最多有 *max_workers* 個行程的池來非同步地執行呼叫。如果 *max_workers* 🄵 `None` 或未給定，它將被預設🄵*os.process_cpu_count()*。如果 *max_workers* 小於或等於 `0`，則會引發*ValueError*。在 Windows 上，*max_workers* 必須小於或等於 `61`。如果不是，則會引發*ValueError*。如果 *max_workers* 🄵 `None`，則預設選擇最多🄵 `61`，即便有更多處理器可用。*mp_context* 可以是*multiprocessing* 情境 (context) 或 `None`。它將用於🄵動 worker。如果 *mp_context* 🄵 `None` 或未給定，則使用預設的*multiprocessing* 情境。請見*Contexts and start methods*。

initializer 是一個可選的可呼叫物件，在每個工作行程 (worker process) 開始時呼叫；*initargs* 是傳遞給 initializer 的引數元組。如果 *initializer* 引發例外，所有目前未定的作業以及任何向池中提交更多作業的嘗試都將引發*BrokenProcessPool*。

max_tasks_per_child 是一個可選引數，它指定單個行程在退出🄵被新的工作行程替🄵之前可以執行的最大任務數。預設情🄵下 *max_tasks_per_child* 是 `None`，這意味著工作行程的生命🄵期將與池一樣長。當指定最大值時，在🄵有 *mp_context* 參數的情🄵下，將預設使用"spawn" 做🄵 multiprocessing 🄵動方法。此功能與"fork" 🄵動方法不相容。

在 3.3 版的變更: 當其中一個工作行程突然終止時，現在會引發*BrokenProcessPool* 錯誤。在過去，此行🄵是未定義的 (undefined)，但對 executor 或其 future 的操作經常會發生凍結或死鎖。

在 3.7 版的變更: 新增了 *mp_context* 引數以允許使用者控制由池所建立的工作行程的 start_method。

新增 *initializer* 與 *initargs* 引數。

在 3.11 版的變更: 新增了 *max_tasks_per_child* 引數以允許使用者控制池中 worker 的生命🄵期。

在 3.12 版的變更: 在 POSIX 系統上，如果你的應用程式有多個執行緒🄵且*multiprocessing* 情境使用了 "fork" 🄵動方法: 🄵部呼叫以🄵生 worker 的*os.fork()* 函式可能會引發*DeprecationWarning*。傳遞一個 *mp_context* 以配置🄵使用不同的🄵動方法。更多🄵明請參🄵*os.fork()* 文件。

在 3.13 版的變更: *max_workers* uses *os.process_cpu_count()* by default, instead of *os.cpu_count()*.

在 3.14 版的變更: The default process start method (see *Contexts and start methods*) changed away from *fork*. If you require the *fork* start method for *ProcessPoolExecutor* you must explicitly pass `mp_context=multiprocessing.get_context("fork")`.

terminate_workers()

Attempt to terminate all living worker processes immediately by calling *Process.terminate* on each of them. Internally, it will also call *Executor.shutdown()* to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

在 3.14 版被加入.

kill_workers()

Attempt to kill all living worker processes immediately by calling *Process.kill* on each of them. Internally, it will also call *Executor.shutdown()* to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

在 3.14 版被加入.

ProcessPoolExecutor 范例

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
```

(繼續下一頁)

(繼續上一頁)

```
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

18.5.5 Future 物件

Future 類🄵封裝了可呼叫物件的非同步執行。*Future* 實例由*Executor.submit()* 建立。

class concurrent.futures.**Future**

封裝可呼叫物件的非同步執行。*Future* 實例由*Executor.submit()* 建立，且除測試外不應直接建立。

cancel()

嘗試取消呼叫。如果呼叫目前正在執行或已完成運行且無法取消，則該方法將回傳 `False`，否則呼叫將被取消🄵且該方法將回傳 `True`。

cancelled()

如果該呼叫成功被取消，則回傳 `True`。

running()

如果呼叫正在執行且無法取消，則回傳 `True`。

done()

如果呼叫成功被取消或結束運行，則回傳 `True`。

result(*timeout=None*)

回傳該呼叫回傳的值。如果呼叫尚未完成，則此方法將等待至多 *timeout* 秒。如果呼叫在 *timeout* 秒🄵未完成，則會引發*TimeoutError*。*timeout* 可以是整數或浮點數。如果未指定 *timeout* 或🄵 `None`，則等待時間就不會有限制。

如果 future 在完成之前被取消，那🄵*CancelledError* 將被引發。

如果該呼叫引發了例外，此方法將引發相同的例外。

exception(*timeout=None*)

回傳該呼叫引發的例外。如果呼叫尚未完成，則此方法將等待至多 *timeout* 秒。如果呼叫在 *timeout* 秒🄵未完成，則會引發*TimeoutError*。*timeout* 可以是整數或浮點數。如果未指定 *timeout* 或🄵 `None`，則等待時間就不會有限制。

如果 future 在完成之前被取消，那🄵*CancelledError* 將被引發。

如果呼叫在🄵有引發的情🄵下完成，則回傳 `None`。

add_done_callback(*fn*)

將可呼叫的 *fn* 附加到 future 上。當 future 被取消或完成運行時，*fn* 將被以 future 作🄵其唯一引數來呼叫。

新增的可呼叫物件按新增順序呼叫，🄵且始終在屬於新增它們的行程的執行緒中呼叫。如果可呼叫物件引發*Exception* 子類🄵，它將被記🄵 (log) 🄵忽略。如果可呼叫物件引發*BaseException* 子類🄵，該行🄵未定義。

如果 future 已經完成或被取消，*fn* 將立即被呼叫。

以下*Future* 方法旨在用於單元測試和*Executor* 實作。

set_running_or_notify_cancel()

此方法只能在與*Future* 關聯的工作被執行之前於*Executor* 實作中呼叫，或者在單元測試中呼叫。

如果該方法回傳 False 則*Future* 已被取消，即*Future.cancel()* 被呼叫🄵回傳 True。任何等待*Future* 完成的執行緒（即透過*as_completed()* 或*wait()*）將被🄵醒。

如果該方法回傳 True 則代表*Future* 未被取消🄵已進入運行狀態，意即呼叫*Future.running()* 將回傳 True。

此方法只能呼叫一次，且不能在呼叫*Future.set_result()* 或*Future.set_exception()* 之後呼叫。

set_result(*result*)

將與*Future* 關聯的工作結果設定🄵 *result*。

此方法只能在*Executor* 實作中和單元測試中使用。

在 3.8 版的變更: 如果*Future* 已經完成，此方法會引發*concurrent.futures.InvalidStateError*。

set_exception(*exception*)

將與*Future* 關聯的工作結果設定🄵*Exception* *exception*。

此方法只能在*Executor* 實作中和單元測試中使用。

在 3.8 版的變更: 如果*Future* 已經完成，此方法會引發*concurrent.futures.InvalidStateError*。

18.5.6 模組函式

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

等待 *fs* 給定的*Future* 實例（可能由不同的*Executor* 實例建立）完成。提供給 *fs* 的重🄵 future 將被🄵除，🄵且只會回傳一次。回傳一個集合的附名二元組 (named 2-tuple of sets)。第一組名🄵 done，包含在等待完成之前完成的 future（已完成或被取消的 future）。第二組名🄵 not_done，包含未完成的 future（未定或運行中的 future）。

timeout 可用於控制回傳前等待的最大秒數。*timeout* 可以是整數或浮點數。如果未指定 *timeout* 或🄵 None，則等待時間就🄵有限制。

return_when 表示此函式應回傳的時間。它必須是以下常數之一：

常數	描述
concurrent.futures.**FIRST_COMPLETED**	當任何 future 完成或被取消時，該函式就會回傳。
concurrent.futures.**FIRST_EXCEPTION**	該函式會在任何 future 透過引發例外而完結時回傳。如果 future 🄵有引發例外，那🄵它等同於*ALL_COMPLETED*。
concurrent.futures.**ALL_COMPLETED**	當所有 future 都完成或被取消時，該函式才會回傳。

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

回傳由 *fs* 給定的*Future* 實例（可能由不同的*Executor* 實例建立）的🄵代器，它在完成時🄵生 future（已完成或被取消的 future）。*fs* 給定的任何重🄵的 future 將只被回傳一次。呼叫*as_completed()* 之前完成的任何 future 將首先🄵生。如果*__next__()* 被呼叫，🄵且在原先呼叫*as_completed()* 的 *timeout* 秒後結果仍不可用，則回傳的🄵代器會引發*TimeoutError*。*timeout* 可以是整數或浮點數。如果未指定 *timeout* 或🄵 None，則等待時間就🄵有限制。

也參考

PEP 3148 -- futures - 非同步地執行運算
描述此功能🄵提出被包含於 Python 標準函式庫中的提案。

18.5.7 例外類🄵

exception concurrent.futures.**CancelledError**

當 future 被取消時引發。

exception concurrent.futures.**TimeoutError**

TimeoutError 的🄵用🄵名，在 future 操作超過給定超時 (timeout) 時引發。

在 3.11 版的變更: 這個類🄵是*TimeoutError* 的🄵名。

exception concurrent.futures.**BrokenExecutor**

衍生自*RuntimeError*，當執行器因某種原因損壞時會引發此例外類🄴，🄵且不能用於提交或執行新任務。

在 3.7 版被加入.

exception concurrent.futures.**InvalidStateError**

目前狀態下不允許的 future 操作被執行時而引發。

在 3.8 版被加入.

exception concurrent.futures.thread.**BrokenThreadPool**

衍生自*BrokenExecutor*，當*ThreadPoolExecutor* 的其中一個 worker 初始化失敗時會引發此例外類🄵。

在 3.7 版被加入.

exception concurrent.futures.interpreter.**BrokenInterpreterPool**

衍生自*BrokenThreadPool*，當*InterpreterPoolExecutor* 的其中一個 worker 初始化失敗時會引發此例外類🄵。

在 3.14 版被加入.

exception concurrent.futures.interpreter.**ExecutionFailed**

Raised from *InterpreterPoolExecutor* when the given initializer fails or from *submit()* when there's an uncaught exception from the submitted task.

在 3.14 版被加入.

exception concurrent.futures.process.**BrokenProcessPool**

衍生自*BrokenExecutor*（以前🄵*RuntimeError*），當*ProcessPoolExecutor* 的其中一個 worker 以不乾🄵的方式終止時將引發此例外類🄵（例如它是從外面被 kill 掉的）。

在 3.3 版被加入.

18.6 concurrent.interpreters --- 在同一個行程中使用多個直譯器

在 3.14 版被加入.

原始碼： Lib/concurrent/interpreters

The concurrent.interpreters module constructs higher-level interfaces on top of the lower level _interpreters module.

The module is primarily meant to provide a basic API for managing interpreters (AKA "subinterpreters") and running things in them. Running mostly involves switching to an interpreter (in the current thread) and calling a function in that execution context.

For concurrency, interpreters themselves (and this module) don't provide much more than isolation, which on its own isn't useful. Actual concurrency is available separately through *threads* See *below*

也參考

InterpreterPoolExecutor
combines threads with interpreters in a familiar interface.

isolating-extensions-howto
how to update an extension module to support multiple interpreters

PEP 554

PEP 734

PEP 684

可用性: not WASI.

This module does not work or is not available on WebAssembly. See *WebAssembly 平台* for more information.

18.6.1 Key details

Before we dive in further, there are a small number of details to keep in mind about using multiple interpreters:

- *isolated*, by default
- no implicit threads
- not all PyPI packages support use in multiple interpreters yet

18.6.2 Introduction

An ”interpreter” is effectively the execution context of the Python runtime. It contains all of the state the runtime needs to execute a program. This includes things like the import state and builtins. (Each thread, even if there's only the main thread, has some extra runtime state, in addition to the current interpreter, related to the current exception and the bytecode eval loop.)

The concept and functionality of the interpreter have been a part of Python since version 2.2, but the feature was only available through the C-API and not well known, and the *isolation* was relatively incomplete until version 3.12.

Multiple Interpreters and Isolation

A Python implementation may support using multiple interpreters in the same process. CPython has this support. Each interpreter is effectively isolated from the others (with a limited number of carefully managed process-global exceptions to the rule).

That isolation is primarily useful as a strong separation between distinct logical components of a program, where you want to have careful control of how those components interact.

備🄵

Interpreters in the same process can technically never be strictly isolated from one another since there are few restrictions on memory access within the same process. The Python runtime makes a best effort at isolation but extension modules may easily violate that. Therefore, do not use multiple interpreters in security-sensitive situations, where they shouldn't have access to each other's data.

Running in an Interpreter

Running in a different interpreter involves switching to it in the current thread and then calling some function. The runtime will execute the function using the current interpreter's state. The `concurrent.interpreters` module provides a basic API for creating and managing interpreters, as well as the switch-and-call operation.

No other threads are automatically started for the operation. There is *a helper* for that though. There is another dedicated helper for calling the builtin `exec()` in an interpreter.

When *exec()* (or *eval()*) are called in an interpreter, they run using the interpreter's `__main__` module as the "globals" namespace. The same is true for functions that aren't associated with any module. This is the same as how scripts invoked from the command-line run in the `__main__` module.

Concurrency and Parallelism

As noted earlier, interpreters do not provide any concurrency on their own. They strictly represent the isolated execution context the runtime will use *in the current thread*. That isolation makes them similar to processes, but they still enjoy in-process efficiency, like threads.

All that said, interpreters do naturally support certain flavors of concurrency. There's a powerful side effect of that isolation. It enables a different approach to concurrency than you can take with async or threads. It's a similar concurrency model to CSP or the actor model, a model which is relatively easy to reason about.

You can take advantage of that concurrency model in a single thread, switching back and forth between interpreters, Stackless-style. However, this model is more useful when you combine interpreters with multiple threads. This mostly involves starting a new thread, where you switch to another interpreter and run what you want there.

Each actual thread in Python, even if you're only running in the main thread, has its own *current* execution context. Multiple threads can use the same interpreter or different ones.

At a high level, you can think of the combination of threads and interpreters as threads with opt-in sharing.

As a significant bonus, interpreters are sufficiently isolated that they do not share the *GIL*, which means combining threads with multiple interpreters enables full multi-core parallelism. (This has been the case since Python 3.12.)

直譯器之間的溝通

In practice, multiple interpreters are useful only if we have a way to communicate between them. This usually involves some form of message passing, but can even mean sharing data in some carefully managed way.

With this in mind, the `concurrent.interpreters` module provides a *queue.Queue* implementation, available through *create_queue()*.

"Sharing" Objects

Any data actually shared between interpreters loses the thread-safety provided by the *GIL*. There are various options for dealing with this in extension modules. However, from Python code the lack of thread-safety means objects can't actually be shared, with a few exceptions. Instead, a copy must be created, which means mutable objects won't stay in sync.

By default, most objects are copied with *pickle* when they are passed to another interpreter. Nearly all of the immutable builtin objects are either directly shared or copied efficiently. For example:

- *None*
- *bool* (*True* 和*False*)
- *bytes*
- *str*
- *int*
- *float*
- *tuple* (of similarly supported objects)

There is a small number of Python types that actually share mutable data between interpreters:

- *memoryview*
- *Queue*

18.6.3 Reference

This module defines the following functions:

`concurrent.interpreters.`**`list_all`**`()`

Return a *`list`* of *`Interpreter`* objects, one for each existing interpreter.

`concurrent.interpreters.`**`get_current`**`()`

Return an *`Interpreter`* object for the currently running interpreter.

`concurrent.interpreters.`**`get_main`**`()`

Return an *`Interpreter`* object for the main interpreter. This is the interpreter the runtime created to run the *REPL* or the script given at the command-line. It is usually the only one.

`concurrent.interpreters.`**`create`**`()`

Initialize a new (idle) Python interpreter and return a *`Interpreter`* object for it.

`concurrent.interpreters.`**`create_queue`**`()`

Initialize a new cross-interpreter queue and return a *`Queue`* object for it.

Interpreter objects

`class` `concurrent.interpreters.`**`Interpreter`**`(`*id*`)`

A single interpreter in the current process.

Generally, *`Interpreter`* shouldn't be called directly. Instead, use *`create()`* or one of the other module functions.

`id`

(read-only)

The underlying interpreter's ID.

`whence`

(read-only)

A string describing where the interpreter came from.

`is_running``()`

Return `True` if the interpreter is currently executing code in its `__main__` module and `False` otherwise.

`close``()`

Finalize and destroy the interpreter.

`prepare_main``(`*ns=None*, ***kwargs*`)`

Bind objects in the interpreter's `__main__` module.

Some objects are actually shared and some are copied efficiently, but most are copied via *`pickle`*. See *"Sharing" Objects*.

`exec``(`*code*, */*, *dedent=True*`)`

Run the given source code in the interpreter (in the current thread).

`call``(`*callable*, */*, **args*, ***kwargs*`)`

Return the result of calling running the given function in the interpreter (in the current thread).

`call_in_thread``(`*callable*, */*, **args*, ***kwargs*`)`

Run the given function in the interpreter (in a new thread).

例外

`exception` `concurrent.interpreters.`**`InterpreterError`**

This exception, a subclass of *`Exception`*, is raised when an interpreter-related error happens.

exception concurrent.interpreters.**InterpreterNotFoundError**

This exception, a subclass of *InterpreterError*, is raised when the targeted interpreter no longer exists.

exception concurrent.interpreters.**ExecutionFailed**

This exception, a subclass of *InterpreterError*, is raised when the running code raised an uncaught exception.

excinfo

A basic snapshot of the exception raised in the other interpreter.

exception concurrent.interpreters.**NotShareableError**

This exception, a subclass of *TypeError*, is raised when an object cannot be sent to another interpreter.

在直譯器之間溝通

class concurrent.interpreters.**Queue**(*id*)

A wrapper around a low-level, cross-interpreter queue, which implements the *queue.Queue* interface. The underlying queue can only be created through *create_queue()*.

Some objects are actually shared and some are copied efficiently, but most are copied via *pickle*. See *"Sharing" Objects*.

id

(read-only)

The queue's ID.

exception concurrent.interpreters.**QueueEmptyError**

This exception, a subclass of *queue.Empty*, is raised from `Queue.get()` and `Queue.get_nowait()` when the queue is empty.

exception concurrent.interpreters.**QueueFullError**

This exception, a subclass of *queue.Full*, is raised from `Queue.put()` and `Queue.put_nowait()` when the queue is full.

18.6.4 基本用法

建立一個直譯器🄵在其中運行程式碼:

```
from concurrent import interpreters

interp = interpreters.create()

# Run in the current OS thread.

interp.exec('print("spam!")')

interp.exec("""if True:
    print('spam!')
    """)

from textwrap import dedent
interp.exec(dedent("""
    print('spam!')
    """))

def run(arg):
    return arg

res = interp.call(run, 'spam!')
print(res)
```

(繼續下一頁)

(繼續上一頁)

```
def run():
    print('spam!')

interp.call(run)

# Run in new OS thread.

t = interp.call_in_thread(run)
t.join()
```

18.7 subprocess --- 子行程管理

原始碼： Lib/subprocess.py

The *subprocess* module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the *subprocess* module can be used to replace these modules and functions can be found in the following sections.

> **也參考**
>
> **PEP 324** -- 提議 subprocess 模組的 PEP

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

18.7.1 Using the subprocess Module

The recommended approach to invoking subprocesses is to use the *run()* function for all use cases it can handle. For more advanced use cases, the underlying *Popen* interface can be used directly.

subprocess.**run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, **other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return a *CompletedProcess* instance.

The arguments shown above are merely the most common ones, described below in *Frequently Used Arguments* (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the *Popen* constructor - most of the arguments to this function are passed through to that interface. (*timeout*, *input*, *check*, and *capture_output* are not.)

If *capture_output* is true, stdout and stderr will be captured. When used, the internal *Popen* object is automatically created with *stdout* and *stderr* both set to *PIPE*. The *stdout* and *stderr* arguments may not be supplied at the same time as *capture_output*. If you wish to capture and combine both streams into one, set *stdout* to *PIPE* and *stderr* to *STDOUT*, instead of using *capture_output*.

A *timeout* may be specified in seconds, it is internally passed on to *Popen.communicate()*. If the timeout expires, the child process will be killed and waited for. The *TimeoutExpired* exception will be re-raised after the child process has terminated. The initial process creation itself cannot be interrupted on many platform APIs so you are not guaranteed to see a timeout exception until at least after however long process creation takes.

The *input* argument is passed to *Popen.communicate()* and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal *Popen* object is automatically created with *stdin* set to *PIPE*, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a *CalledProcessError* exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the *io.TextIOWrapper* default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to *Popen*. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like *os.environ* or *os.environb*.

範例：

```
>>> subprocess.run(["ls", "-l"])  # 不捕捉輸出
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

在 3.5 版被加入.

在 3.6 版的變更: 新增 *encoding* 與 *errors* 參數。

在 3.7 版的變更: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

在 3.12 版的變更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

The return value from *run()*, representing a process that has finished.

args

The arguments used to launch the process. This may be a list or a string.

returncode

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

stdout

Captured stdout from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. `None` if stdout was not captured.

If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and *stderr* will be `None`.

stderr

Captured stderr from the child process. A bytes sequence, or a string if *run()* was called with an encoding, errors, or text=True. `None` if stderr was not captured.

check_returncode()

If *returncode* is non-zero, raise a *CalledProcessError*.

在 3.5 版被加入.

subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that the special file *os.devnull* will be used.

在 3.3 版被加入.

subprocess.**PIPE**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that a pipe to the standard stream should be opened. Most useful with *Popen.communicate()*.

subprocess.**STDOUT**

Special value that can be used as the *stderr* argument to *Popen* and indicates that standard error should go into the same handle as standard output.

exception subprocess.**SubprocessError**

Base class for all other exceptions from this module.

在 3.3 版被加入.

exception subprocess.**TimeoutExpired**

Subclass of *SubprocessError*, raised when a timeout expires while waiting for a child process.

cmd

Command that was used to spawn the child process.

timeout

Timeout in seconds.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, `None`. This is always *bytes* when any output was captured regardless of the `text=True` setting. It may remain `None` instead of `b''` when no output was observed.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, `None`. This is always *bytes* when stderr output was captured regardless of the `text=True` setting. It may remain `None` instead of `b''` when no stderr output was observed.

在 3.3 版被加入.

在 3.5 版的變更: *stdout* and *stderr* attributes added

exception subprocess.**CalledProcessError**

Subclass of *SubprocessError*, raised when a process run by *check_call()*, *check_output()*, or *run()* (with `check=True`) returns a non-zero exit status.

returncode

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

cmd

Command that was used to spawn the child process.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, `None`.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, None.

在 3.5 版的變更: *stdout* and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the *Popen* constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be *True* (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are None, *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of None, no redirection will occur. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

For *stdin*, line ending characters '\n' in the input will be converted to the default line separator *os.linesep*. For *stdout* and *stderr*, all line endings in the output will be converted to '\n'. For more information see the documentation of the *io.TextIOWrapper* class when the *newline* argument to its constructor is None.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

在 3.6 版的變更: 新增 *encoding* 與 *errors* 參數。

在 3.7 版的變更: Added the *text* parameter as an alias for *universal_newlines*.

> **備🄵**
>
> The newlines attribute of the file objects *Popen.stdin*, *Popen.stdout* and *Popen.stderr* are not updated by the *Popen.communicate()* method.

If *shell* is True, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, and *shutil*).

在 3.3 版的變更: When *universal_newlines* is True, the class uses the encoding *locale.getpreferredencoding(False)* instead of locale.getpreferredencoding(). See the *io.TextIOWrapper* class for more information on this change.

> **備F**
>
> Read the *Security Considerations* section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the `Popen` constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*, *process_group=None*)

Execute a child program in a new process. On POSIX, the class uses `os.execvpe()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.

args should be a sequence of program arguments or else a single string or *path-like object*. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

> **警告**
>
> For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use `shutil.which()`. On all platforms, passing `sys.executable` is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.
>
> Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see `os.execvpe()`, and note that when resolving or searching for the executable path, *cwd* overrides the current working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

> **備F**
>
> It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. `shlex.split()` can illustrate how to determine the correct tokenization for *args*:
>
> ```
> >>> import shlex, subprocess
> >>> command_line = input()
> /bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
> >>> args = shlex.split(command_line)
> ```

```
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # 成功！
```

Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in *Converting an argument sequence to a string on Windows*. This is because the underlying `CreateProcess()` operates on strings.

在 3.6 版的變更: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing path-like objects on POSIX.

在 3.8 版的變更: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing bytes and path-like objects on Windows.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, *Popen* does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **`dir`** or **`copy`**). You do not need `shell=True` to run a batch file or console-based executable.

備🄵

Read the *Security Considerations* section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the *open()* function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `text=True` or `universal_newlines=True`)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

在 3.3.1 版的變更: *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **`ps`**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

在 3.6 版的變更: *executable* parameter accepts a *path-like object* on POSIX.

在 3.8 版的變更: *executable* parameter accepts a bytes and *path-like object* on Windows.

在 3.12 版的變更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of `None`, no redirection will occur. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for *stdout*.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

警告

The *preexec_fn* parameter is NOT SAFE to use in the presence of threads in your application. The child process could deadlock before exec is called.

備🄵

If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* and *process_group* parameters should take the place of code using *preexec_fn* to call `os.setsid()` or `os.setpgid()` in the child.

在 3.8 版的變更: The *preexec_fn* parameter is no longer supported in subinterpreters. The use of the parameter in a subinterpreter raises `RuntimeError`. The new restriction may affect applications that are deployed in mod_wsgi, uWSGI, and other embedded environments.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in *Inheritance of File Descriptors*.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of `STARTUPINFO.lpAttributeList`, or by standard handle redirection.

在 3.2 版的變更: The default for *close_fds* was changed from `False` to what is described above.

在 3.7 版的變更: On Windows the default for *close_fds* was changed from `False` to `True` when redirecting the standard handles. It's now possible to set *close_fds* to `True` when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be `True`. (POSIX only)

在 3.2 版的變更: 新增 *pass_fds* 參數。

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or *path-like* object. On POSIX, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

在 3.6 版的變更: *cwd* parameter accepts a *path-like object* on POSIX.

在 3.7 版的變更: *cwd* parameter accepts a *path-like object* on Windows.

在 3.8 版的變更: *cwd* parameter accepts a bytes object on Windows.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

在 3.2 版的變更: 新增 *restore_signals*。

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess.

可用性: POSIX

在 3.2 版的變更: 新增 *start_new_session*。

If *process_group* is a non-negative integer, the `setpgid(0, value)` system call will be made in the child process prior to the execution of the subprocess.

可用性: POSIX

在 3.11 版的變更: 新增 *process_group*。

If *group* is not `None`, the setregid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via *grp.getgrnam()* and the value in `gr_gid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

可用性: POSIX

在 3.9 版被加入.

If *extra_groups* is not `None`, the setgroups() system call will be made in the child process prior to the execution of the subprocess. Strings provided in *extra_groups* will be looked up via *grp.getgrnam()* and the values in `gr_gid` will be used. Integer values will be passed verbatim. (POSIX only)

可用性: POSIX

在 3.9 版被加入.

If *user* is not `None`, the setreuid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via *pwd.getpwnam()* and the value in `pw_uid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

可用性: POSIX

在 3.9 版被加入.

If *umask* is not negative, the umask() system call will be made in the child process prior to the execution of the subprocess.

可用性: POSIX

在 3.9 版被加入.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like *os.environ* or *os.environb*.

備🄵

If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified *encoding* and *errors*, as described above in *Frequently Used Arguments*. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

在 3.6 版被加入: 新增 *encoding* 與 *errors*。

在 3.7 版被加入: *text* was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a *STARTUPINFO* object, which is passed to the underlying `CreateProcess` function.

If given, *creationflags*, can be one or more of the following flags:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

pipesize can be used to change the size of the pipe when *PIPE* is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

在 3.10 版的變更: 新增 `pipesize` 參數。

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen and the other functions in this module that use it raise an *auditing event* `subprocess.Popen` with arguments `executable`, `args`, `cwd`, and `env`. The value for `args` may be a single string or a list of strings, depending on platform.

在 3.2 版的變更: 新增情境管理器的支援。

在 3.6 版的變更: Popen destructor now emits a *ResourceWarning* warning if the child process is still running.

在 3.8 版的變更: Popen can use *os.posix_spawn()* in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, Popen constructor using *os.posix_spawn()* no longer raise an exception on errors like missing program, but the child process fails with a non-zero *returncode*.

例外

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

The most common exception raised is *OSError*. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for *OSError* exceptions. Note that, when `shell=True`, *OSError* will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

A *ValueError* will be raised if *Popen* is called with invalid arguments.

check_call() and *check_output()* will raise *CalledProcessError* if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as *run()* and *Popen.communicate()* will raise *TimeoutExpired* if the timeout expires before the process exits.

Exceptions defined in this module all inherit from *SubprocessError*.

在 3.3 版被加入: The *SubprocessError* base class was added.

18.7.2 安全性注意事項

Unlike some other popen functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. On *some platforms*, it is possible to use *`shlex.quote()`* for this escaping.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 for additional discussion.

18.7.3 Popen Objects

Instances of the *`Popen`* class have the following methods:

`Popen.`**`poll`**`()`

Check if child process has terminated. Set and return *`returncode`* attribute. Otherwise, returns `None`.

`Popen.`**`wait`**`(`*timeout=None*`)`

Wait for child process to terminate. Set and return *`returncode`* attribute.

If the process does not terminate after *timeout* seconds, raise a *`TimeoutExpired`* exception. It is safe to catch this exception and retry the wait.

> **備🄵**
>
> This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use *`Popen.communicate()`* when using pipes to avoid that.

> **備🄵**
>
> When the `timeout` parameter is not `None`, then (on POSIX) the function is implemented using a busy loop (non-blocking call and short sleeps). Use the *`asyncio`* module for an asynchronous wait: see *`asyncio.create_subprocess_exec`*.

在 3.3 版的變更: 新增 *timeout*。

`Popen.`**`communicate`**`(`*input=None*, *timeout=None*`)`

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the *`returncode`* attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a *`TimeoutExpired`* exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **備🄵**
>
> The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

在 3.3 版的變更: 新增 *timeout*。

`Popen.`**`send_signal`**`(`*signal*`)`

Sends the signal *signal* to the child.

Do nothing if the process completed.

> **備🄵**
>
> On Windows, SIGTERM is an alias for *`terminate()`*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

`Popen.`**`terminate`**`()`

Stop the child. On POSIX OSs the method sends *`SIGTERM`* to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

`Popen.`**`kill`**`()`

Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows *`kill()`* is an alias for *`terminate()`*.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:

`Popen.`**`args`**

The *args* argument as it was passed to *`Popen`* -- a sequence of program arguments or else a single string.

在 3.3 版被加入.

`Popen.`**`stdin`**

If the *stdin* argument was *`PIPE`*, this attribute is a writeable stream object as returned by *`open()`*. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *`PIPE`*, this attribute is `None`.

`Popen.`**`stdout`**

If the *stdout* argument was *`PIPE`*, this attribute is a readable stream object as returned by *`open()`*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *`PIPE`*, this attribute is `None`.

`Popen.`**`stderr`**

If the *stderr* argument was *`PIPE`*, this attribute is a readable stream object as returned by *`open()`*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *`PIPE`*, this attribute is `None`.

> **警告**
>
> Use *communicate()* rather than *.stdin.write*, *.stdout.read* or *.stderr.read* to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

`Popen.`**`pid`**

The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

`Popen.`**`returncode`**

The child return code. Initially `None`, *returncode* is set by a call to the *poll()*, *wait()*, or *communicate()* methods if they detect that the process has terminated.

A `None` value indicates that the process hadn't yet terminated at the time of the last method call.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

18.7.4 Windows Popen Helpers

The *STARTUPINFO* class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**(**, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None*)

Partial support of the Windows STARTUPINFO structure is used for *Popen* creation. The following attributes can be set by passing them as keyword-only arguments.

在 3.7 版的變更: Keyword-only argument support was added.

dwFlags

A bit field that determines whether certain *STARTUPINFO* attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard input handle for the process. If *STARTF_USESTDHANDLES* is not specified, the default for standard input is the keyboard buffer.

hStdOutput

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

hStdError

If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

wShowWindow

If *dwFlags* specifies *STARTF_USESHOWWINDOW*, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

SW_HIDE is provided for this attribute. It is used when *Popen* is called with `shell=True`.

lpAttributeList

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see UpdateProcThreadAttribute.

Supported attributes:

handle_list
Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by `os.set_handle_inheritable()` when passed to the `Popen` constructor, else `OSError` will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

> **⚠ 警告**
>
> In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as `os.system()`. This also applies to standard handle redirection, which temporarily creates inheritable handles.

在 3.7 版被加入.

Windows Constants

The `subprocess` module exposes the following constants.

`subprocess.`**`STD_INPUT_HANDLE`**
The standard input device. Initially, this is the console input buffer, `CONIN$`.

`subprocess.`**`STD_OUTPUT_HANDLE`**
The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.`**`STD_ERROR_HANDLE`**
The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.`**`SW_HIDE`**
Hides the window. Another window will be activated.

`subprocess.`**`STARTF_USESTDHANDLES`**
Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

`subprocess.`**`STARTF_USESHOWWINDOW`**
Specifies that the `STARTUPINFO.wShowWindow` attribute contains additional information.

`subprocess.`**`STARTF_FORCEONFEEDBACK`**
A `STARTUPINFO.dwFlags` parameter to specify that the *Working in Background* mouse cursor will be displayed while a process is launching. This is the default behavior for GUI processes.

在 3.13 版被加入.

`subprocess.`**`STARTF_FORCEOFFFEEDBACK`**
A `STARTUPINFO.dwFlags` parameter to specify that the mouse cursor will not be changed when launching a process.

在 3.13 版被加入.

`subprocess.`**`CREATE_NEW_CONSOLE`**
The new process has a new console, instead of inheriting its parent's console (the default).

`subprocess.`**`CREATE_NEW_PROCESS_GROUP`**
A `Popen` `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

This flag is ignored if `CREATE_NEW_CONSOLE` is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have an above average priority.

在 3.7 版被加入.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have a below average priority.

在 3.7 版被加入.

subprocess.**HIGH_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have a high priority.

在 3.7 版被加入.

subprocess.**IDLE_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

在 3.7 版被加入.

subprocess.**NORMAL_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have a normal priority. (default)

在 3.7 版被加入.

subprocess.**REALTIME_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

在 3.7 版被加入.

subprocess.**CREATE_NO_WINDOW**

A *Popen* `creationflags` parameter to specify that a new process will not create a window.

在 3.7 版被加入.

subprocess.**DETACHED_PROCESS**

A *Popen* `creationflags` parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

在 3.7 版被加入.

subprocess.**CREATE_DEFAULT_ERROR_MODE**

A *Popen* `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

在 3.7 版被加入.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

A *Popen* `creationflags` parameter to specify that a new process is not associated with the job.

在 3.7 版被加入.

18.7.5 Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use *run()* in many cases, but lots of existing code calls these functions.

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

```
run(...).returncode
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **備F**
>
> Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

在 3.3 版的變更: 新增 *timeout*。

在 3.12 版的變更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.**check_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute. If *check_call()* was unable to start the process it will propagate the exception that was raised.

Code needing to capture stdout or stderr should use *run()* instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **備F**
>
> Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

在 3.3 版的變更: 新增 *timeout*。

在 3.12 版的變更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.**check_output**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

Run command with arguments and return its output.

If the return code was non-zero it raises a *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute and any output in the *output* attribute.

這等同於:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. One API deviation from *run()* behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in *Frequently Used Arguments* and *run()*.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

在 3.1 版被加入.

在 3.3 版的變更: 新增 *timeout*。

在 3.4 版的變更: 新增 *input* 關鍵字引數的支援。

在 3.6 版的變更: 新增 *encoding* 與 *errors*。細節請見*run()*。

在 3.7 版被加入: *text* was added as a more readable alias for *universal_newlines*.

在 3.12 版的變更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

18.7.6 Replacing Older Functions with the `subprocess` Module

In this section, "a becomes b" means that b can be used as a replacement for a.

> **備🄵**
>
> All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise *OSError* instead.
>
> In addition, the replacements using *check_output()* will fail with a *CalledProcessError* if the requested operation produces a non-zero return code. The output is still available as the *output* attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the *subprocess* module.

Replacing `/bin/sh` shell command substitution

```
output=$(mycmd myarg)
```

變成:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=$(dmesg | grep hda)
```

變成：

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The `p1.stdout.close()` call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=$(dmesg | grep hda)
```

變成：

```
output = check_output("dmesg | grep hda", shell=True)
```

取代 `os.system()`

```
sts = os.system("mycmd" + " myarg")
# 變成
retcode = call("mycmd" + " myarg", shell=True)
```

🄵解：

- Calling the program through the shell is usually not required.
- The *`call()`* return value is encoded differently to that of *`os.system()`*.
- The *`os.system()`* function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the *`subprocess`* module.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

P_NOWAIT 範例：

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT 範例：

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

向量範例：

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

環境範例：

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

取代 `os.popen()`

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

18.7.7 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.`**`getstatusoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

Return `(exitcode, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. *encoding* and *errors* are used to decode output; see the notes on *Frequently Used Arguments* for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

可用性: Unix, Windows.

在 3.3.4 版的變更: 新增對 Windows 的支援。

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as *`returncode`*.

在 3.11 版的變更: 新增 *encoding* 與 *errors* 參數。

`subprocess.`**`getoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

Return output (stdout and stderr) of executing *cmd* in a shell.

Like *getstatusoutput()*, except the exit code is ignored and the return value is a string containing the command's output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

可用性: Unix, Windows.

在 3.3.4 版的變更: 新增對 Windows 的支援

在 3.11 版的變更: 新增 *encoding* 與 *errors* 參數。

18.7.8 🄵解

Timeout Behavior

When using the `timeout` parameter in functions like *run()*, *Popen.wait()*, or *Popen.communicate()*, users should be aware of the following behaviors:

1. **Process Creation Delay**: The initial process creation itself cannot be interrupted on many platform APIs. This means that even when specifying a timeout, you are not guaranteed to see a timeout exception until at least after however long process creation takes.
2. **Extremely Small Timeout Values**: Setting very small timeout values (such as a few milliseconds) may result in almost immediate *TimeoutExpired* exceptions because process creation and system scheduling inherently require time.

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

也參考

shlex
Module which provides function to parse and escape command lines.

停用 `posix_spawn()`

在 Linux 上，*subprocess* 在安全的情🄵下預設會在🄵部使用 `vfork()` 系統呼叫，而不是 `fork()`，這顯著地提高了性能。

```
subprocess._USE_POSIX_SPAWN = False  # 見 CPython 問題 gh-NNNNNN。
```

在任何 Python 版本上將這個設定🄵 false 都是安全的。當不受支援時，它對舊版本或🄵有支援的新版本🄵有影響。不要假設屬性可供讀取。🄵管它的名稱是這樣，真實值🄵不表示將使用相應的函式，而只是表示可能會使用。

每當你需要使用這些私有開關以重現你所看到的問題時，請隨時提出問題 (file issues)。從程式碼中的🄵解連結到該問題。

在 3.8 版被加入: _USE_POSIX_SPAWN

18.8 sched --- 事件排程器

原始碼： Lib/sched.py

---

sched 模組定義了一個有實作通用事件排程器的類🄵：

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

scheduler 類🄵定義了事件排程的泛用介面。它需要兩個函式來和「外部世界」作用——*timefunc* 應不帶引數地被呼叫，🄵回傳一個數字（🄵「時間」，可🄵任何單位）。*delayfunc* 函式應以一個引數呼叫，🄵與 *timefunc* 的輸出相容，且應延遲其指定的時間單位。每個事件運行後，也會以引數 0 呼叫 *delayfunc*，來使得其他執行緒有機會在多執行緒應用程式中運行。

在 3.3 版的變更: *timefunc* 和 *delayfunc* 參數是可選的。

在 3.3 版的變更: *scheduler* 類🄵可以安全地在多執行緒環境中使用。

範例：

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # 🄵管優先順序較高，但由於 enter() 是相對的，'keyword' 在 'positional' 之後運行
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

18.8.1 排程器物件

scheduler 實例具有以下方法和屬性：

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

🄵一個新事件排程。*time* 引數應該是與傳遞給建構函式的 *timefunc* 函式回傳值相容的數字型🄵。安排在相同 *time* 的事件將按照其 *priority* 的順序執行。數字越小代表優先順序越高。

執行事件意味著執行 `action(*argument, **kwargs)`。*argument* 是一個包含 *action* 之位置引數的序列。*kwargs* 是一個字典，帶有 *action* 的關鍵字引數。

回傳值是一個事件，可用於後續取消該事件（請見*cancel()*）。

在 3.3 版的變更: *argument* 參數是可選的。

在 3.3 版的變更: 新增 *kwargs* 參數。

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

🄵一個排程事件延期 *delay* 個時間單位。除了相對時間之外，其他引數、效果和回傳值皆與*enterabs()* 的相同。

在 3.3 版的變更: *argument* 參數是可選的。

在 3.3 版的變更: 新增 *kwargs* 參數。

scheduler.**cancel**(*event*)

從🄵列中🄵除該事件。如果 *event* 不是目前🄵列中的事件，此方法將引發*ValueError*。

scheduler.**empty**()

如果事件🄵列🄵空，則回傳 `True`。

scheduler.**run**(*blocking=True*)

運行所有已排程的事件。此方法將會等待（使用傳遞給建構函式的 *delayfunc* 函式）下一個事件，然後執行它，🄵依此類推，直到不再有排程好的事件。

如果 *blocking* 🄵 false，則執行最快到期的已排程事件（如存在），然後回傳排程器中下一個排程呼叫（如存在）的截止時間。

action 或 *delayfunc* 都可能引發例外。無論哪種情🄵，排程器都將保持一致的狀態🄵傳遞例外。如果是由 *action* 引發例外，則後續呼叫*run()* 時將不會嘗試運行該事件。

如果一系列事件的運行時長比執行下一個事件前的可用時長 (available time) 更長，那🄵排程器就單純會落後。不會有事件被🄵🄵；呼叫程式碼也要負責取消不再相關的事件。

在 3.3 版的變更: 新增 *blocking* 參數。

scheduler.**queue**

會按事件運行順序回傳即將發生的事件串列的唯讀屬性。每個事件都以附 名元組 *(named tuple)* 表示，🄵包含以下欄位：時間、優先順序、動作 (action)、引數、kwargs。

18.9 queue --- 同步🄵列 (synchronized queue) 類🄵

原始碼： Lib/queue.py

---

queue module（模組）實作多生🄵者、多消費者🄵列。在執行緒程式設計中，必須在多執行緒之間安全地交🄵資訊時，特🄵有用。此 module 中的*Queue* class 實作所有必需的鎖定語義 (locking semantics)。

此 module 實作三種型🄵的🄵列，它們僅在取出條目的順序上有所不同。在 FIFO 🄵列中，先加入的任務是第一個被取出的。在 LIFO 🄵列中，最近被加入的條目是第一個被取出的（像堆🄵 (stack) 一樣操作）。使用優先🄵列 (priority queue) 時，條目將保持排序狀態（使用*heapq* module），🄵先取出最低值條目。

在🄵部，這三種型🄵的🄵列使用鎖 (lock) 來暫時阻塞競🄵執行緒；但是，它們🄵不是被設計來處理執行緒🄵的 reentrancy（可重入）。

此外，此 module 實作一個「簡單」的 FIFO 🄵列型🄵*SimpleQueue*，其特定的實作是以較少的功能🄵代價，來提供額外的保證。

queue module 定義了以下的 class 和例外：

class queue.**Queue**(*maxsize=0*)

FIFO 🄵列的建構子 (constructor)。*maxsize* 是一個整數，用於設置🄵列中可放置的項目數的上限。一旦達到此大小，插入將會阻塞，直到🄵列中的項目被消耗。如果 *maxsize* 小於或等於零，則🄵列大小🄵無限。

class queue.**LifoQueue**(*maxsize=0*)

LIFO 🄵列的建構子。*maxsize* 是一個整數，用於設置🄵列中可放置的項目數的上限。一旦達到此大小，插入將被鎖定，到🄵列中的項目被消耗。如果 *maxsize* 小於或等於零，則🄵列大小🄵無限。

class queue.**PriorityQueue**(*maxsize=0*)

優先🄵列的建構子。*maxsize* 是一個整數，用於設置🄵列中可放置的項目數的上限。一旦達到此大小，插入將被阻塞，直到🄵列中的項目被消耗。如果 *maxsize* 小於或等於零，則🄵列大小🄵無限。

最低值的條目會最先被取出（最低值的條目是被會 `min(entries)` 回傳的那一個）。條目的典型模式是格式🄵 `(priority_number, data)` 的 tuple（元組）。

如果 *data* 元素🄵不可比較的，則可以將資料包裝在一個 class 中，該 class 忽略資料項目🄵僅比較優先數：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**

無界的 FIFO 🄵列的建構子。簡單🄵列缺少任務追🄵等進階功能。

在 3.7 版被加入.

exception queue.**Empty**

當對一個空的*Queue* 物件呼叫非阻塞的 (non-blocking) *get()*（或*get_nowait()*）將引發此例外。

exception queue.**Full**

當對一個已滿的*Queue* 物件呼叫非阻塞的*put()*（或*put_nowait()*）將引發此例外。

exception queue.**ShutDown**

當對一個已關閉的*Queue* 物件呼叫*put()* 或*get()* 時將引發此例外。

在 3.13 版被加入.

18.9.1 🄵列物件

🄵列物件（*Queue*、*LifoQueue*、*PriorityQueue*）提供下面描述的公用 method。

Queue.**qsize**()

回傳🄵列的近似大小。注意，qsize() > 0 🄵不能保證後續的 get() 不會阻塞，qsize() < maxsize 也不會保證 put() 不會阻塞。

Queue.**empty**()

如果🄵列🄵空，則回傳 `True`，否則回傳 `False`。如果 empty() 回傳 `True`，則不保證後續呼叫 put() 不會阻塞。同樣，如果 empty() 回傳 `False`，則不保證後續呼叫 get() 不會阻塞。

Queue.**full**()

如果🄵列已滿，則回傳 `True` ，否則回傳 `False`。如果 full() 回傳 `True`，則不保證後續呼叫 get() 不會阻塞。同樣，如果 full() 回傳 `False`，則不保證後續呼叫 put() 不會阻塞。

Queue.**put**(*item*, *block=True*, *timeout=None*)

將 *item* 放入🄵列中。如果可選的 args *block* 🄵 true、*timeout* 🄵 `None`（預設值），則在必要時阻塞，直到自由槽 (free slot) 可用。如果 *timeout* 🄵正數，則最多阻塞 *timeout* 秒，如果該時間🄵🄵有可用的自由槽，則會引發*Full* 例外。否則（*block* 🄵 false），如果自由槽立即可用，則將項目放在🄵列中，否則引發*Full* 例外（在這種情🄵下，*timeout* 將被忽略）。

如果🄵列已被關閉，則引發*ShutDown*。

Queue.**put_nowait**(*item*)

等效於 `put(item, block=False)`。

Queue.**get**(*block=True*, *timeout=None*)

從🄵列中移除🄵回傳一個項目。如果可選的 args *block* 🄵 true，且 *timeout* 🄵 None（預設值），則在必要時阻塞，直到有可用的項目。如果 *timeout* 是正數，則最多會阻塞 *timeout* 秒，如果該時間🄵🄵有可用的項目，則會引發*Empty* 例外。否則（*block* 🄵 false），如果立即可用，則回傳一個項目，否則引發*Empty* 例外（在這種情🄵下，*timeout* 將被忽略）。

在 POSIX 系統的 3.0 版之前，以及 Windows 的所有版本，如果 *block* 🄵 true 且 *timeout* 🄵 None，則此操作將在底層鎖上進入不間斷等待。這意味著不會發生例外，特🄵是 SIGINT（中斷訊號）不會觸發*KeyboardInterrupt*。

如果🄵列已被關閉且🄵空，或🄵列已立即關閉，則引發*ShutDown*。

Queue.**get_nowait**()

等效於 get(False)。

有兩個 method 可以支援追🄵放入🄵列的任務是否已由常駐消費者執行緒 (daemon consumer threads) 完全處理。

Queue.**task_done**()

表示先前放入🄵列的任務已完成。由🄵列消費者執行緒使用。對於用來提取任務的每個*get()*，隨後呼叫*task_done()* 告訴🄵列任務的處理已完成。

如果目前*join()* 阻塞，它將會在所有項目都已處理完畢後恢復（代表對於以*put()* 放進🄵列的每個項目，都要收到*task_done()* 的呼叫）。

shutdown(immediate=True) calls *task_done()* for each remaining item in the queue.

如果呼叫次數超過🄵列中放置的項目數量，則引發*ValueError*。

Queue.**join**()

持續阻塞直到🄵列中的所有項目都已被取得🄵處理完畢。

每當項目被加到🄵列中時，未完成任務的計數都會增加。每當消費者執行緒呼叫*task_done()* 以指示該項目已被取出🄵且對其的所有工作都已完成時，計數就會下降。當未完成任務的計數降至零時，*join()* 將停止阻塞。

如何等待放入🄵列的任務完成的範例：

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# Turn-on the worker thread.
threading.Thread(target=worker, daemon=True).start()

# Send thirty task requests to the worker.
for item in range(30):
    q.put(item)

# Block until all tasks are done.
q.join()
print('All work completed')
```

Terminating queues

Queue objects can be made to prevent further interaction by shutting them down.

Queue.**shutdown**(*immediate=False*)

Shut down the queue, making *get()* and *put()* raise *ShutDown*.

By default, *get()* on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make *get()* raise immediately instead.

All blocked callers of *put()* and *get()* will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of *join()*.

在 3.13 版被加入.

18.9.2 SimpleQueue 物件

SimpleQueue 物件提供下面描述的公用 method。

SimpleQueue.**qsize**()

傳回F列的近似大小。注意，qsize() > 0 F不能保證後續的 get() 不會阻塞。

SimpleQueue.**empty**()

如果F列F空，則回傳 `True`，否則回傳 `False`。如果 empty() 回傳 `False`，則不保證後續呼叫 get() 不會阻塞。

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)

將 *item* 放入F列中。此 method 從不阻塞，F且都會成功（除了F在的低階錯誤，像是分配記憶體失敗）。可選的 args *block* 和 *timeout* 會被忽略，它們僅是F了與*Queue.put()* 相容才存在。

此 method 有一個可重入 (reentrant) 的 C 實作。意思就是，一個 `put()` 或 `get()` 呼叫，可以被同一執行緒中的另一個 `put()` 呼叫中斷，而不會造成死鎖 (deadlock) 或損壞F列中的F部狀態。這使得它適合在解構子 (destructor) 中使用，像是 `__del__` method 或*weakref* 回呼函式 (callback)。

SimpleQueue.**put_nowait**(*item*)

等效於 `put(item, block=False)`，用於與*Queue.put_nowait()* 相容。

SimpleQueue.**get**(*block=True*, *timeout=None*)

從F列中移除F回傳一個項目。如果可選的 args *block* F true，且 *timeout* F `None`（預設值），則在必要時阻塞，直到有可用的項目。如果 *timeout* 是正數，則最多會阻塞 *timeout* 秒，如果該時間FF有可用的項目，則會引發*Empty* 例外。否則（*block* F false），如果立即可用，則回傳一個項目，否則引發*Empty* 例外（在這種情F下，*timeout* 將被忽略）。

SimpleQueue.**get_nowait**()

等效於 `get(False)`。

也參考

Class ***multiprocessing.Queue***
用於多行程處理 (multi-processing)（而非多執行緒）情境 (context) 的F列 class。

collections.deque 是無界F列的替代實作，有快速且具原子性 (atomic) 的*append()* 和*popleft()* 操作，這些操作不需要鎖定，F且還支持索引。

18.10 contextvars --- 情境變數

本模組供 API 來管理、儲存及存取單一情境各自的狀態（context-local state）。用*ContextVar* 類F宣告和處理情境變數。*copy_context()* 函式和*Context* 類F應在非同步框架中管理目前的情境。

帶有狀態的 Context Manager 應該使用情境變數，而不是*threading.local()*，才能防止它們的狀態在🄵行（concurrent）程式碼中使用時意外外溢🄵干擾到其他程式碼。

其他詳細資訊，請參🄵 **PEP 567**。

在 3.7 版被加入.

18.10.1 情境變數

class contextvars.**ContextVar**(*name*[, *, *default*])

此類🄵用在宣告新的情境變數，例如：

```
var: ContextVar[int] = ContextVar('var', default=42)
```

必要參數 *name* 用於自我檢查（introspection）和除錯。

當在目前的情境中找不到變數的值時，*ContextVar.get()* 會回傳可選的僅限關鍵字參數 *default*。

重要： 情境變數應該在最頂端的模組層級建立，🄵對不要在閉包（closure）中建立。*Context* 物件持有情境變數的🄵參照，這會阻止情境變數被正確地垃圾回收（garbage collected）。

name

這個變數的名稱。這是一個唯讀屬性。

在 3.7.1 版被加入.

get([*default*])

回傳當前情境的情境變數值。

如果在當前情境中🄵有變數的值，此方法將：

- 回傳方法的 *default* 引數值（如果有的話）；否則
- 回傳情境變數的預設值（如果建立情境變數時有指定預設值的話）；否則
- 會引發一個*LookupError*。

set(*value*)

在目前的情境中，呼叫以設定情境變數的新值。

value 屬必要引數，是情境變數的新值。

回傳一個*Token* 物件，該物件可透過*ContextVar.reset()* 方法，用來將變數還原到之前的值。

reset(*token*)

將情境變數重設🄵使用*ContextVar.set()* 建立 *token* 前的值。

舉例來🄵：

```
var = ContextVar('var')

token = var.set('new value')
# 使用 'var' 的程式碼；var.get() 回傳 'new value'。
var.reset(token)

# 在重設呼叫之後，var 又🄵有值了，所以
# var.get() 會引發 LookupError。
```

class contextvars.**Token**

Token 物件由*ContextVar.set()* 方法回傳，可以傳遞給*ContextVar.reset()* 方法，用以將變數的值還原🄵相對應的 *set* 之前的值。

The token supports context manager protocol to restore the corresponding context variable value at the exit from `with` block:

```
var = ContextVar('var', default='default value')

with var.set('new value'):
    assert var.get() == 'new value'

assert var.get() == 'default value'
```

在 3.14 版被加入: 新增對用作情境管理器的支援。

var

唯讀屬性。指向建立 token 的*ContextVar* 物件。

old_value

唯讀屬性。值🄵變數在呼叫*ContextVar.set()* 方法之前的值。如果變數在呼叫前🄵有設定，則指向*Token.MISSING*。

MISSING

由*Token.old_value* 使用的標記物件。

18.10.2 手動情境管理

contextvars.**copy_context**()

回傳目前*Context* 物件的🄵本（copy）。

以下程式碼片段會取得目前情境的🄵本，🄵顯示在其中設定的所有變數及其值：：

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

這個函式具有 *O*(1) 的🄵雜度，也就是🄵，對於只有少許情境變數的情境和有大量情境變數的情境，速度都一樣快。

class contextvars.**Context**

ContextVars 到其值的映射。

`Context()` 會建立一個🄵有值的空情境。要取得目前情境的🄵本，請使用*copy_context()* 函式。

每個執行緒都有自己的 `Context` 物件中目前主控中的堆🄵（stack）。*current context* 是目前執行緒堆🄵頂端的 `Context` 物件。堆🄵中的所有 `Context` 物件都被視🄵已進入。

進入一個情境，可以藉由呼叫其*run()* 方法來完成，此進入的動作會將一情境推到目前執行緒的情境堆🄵的頂端，使該情境成🄵目前的情境。

如果你傳遞給*run()* 方法的回呼函式（callback functions），該函式回傳之後，就會自動退出目前的情境，這會將目前的情境還原到進入情境之前的狀態，方法是將情境從情境堆🄵的頂端彈出。

因🄵每個執行緒都有自己的情境堆🄵，當值在不同的執行緒中被指定時，*ContextVar* 物件的行🄵與*threading.local()* 相似。

嘗試進入已進入的情境，包括在其他執行緒中進入的情境，會引發*RuntimeError*。

退出情境後，之後可以重新進入（從任何執行緒）。

任何透過*ContextVar.set()* 方法對*ContextVar* 值的改變都會記🄵在目前的情境中。*ContextVar.get()* 方法回傳與當前情境相關的值。退出情境實際造成的效果會像是將其在進入情境時對情境變數所做的任何變一一彈出🄵還原（如果需要，可以透過重新進入情境來還原值）。

情境（Context）實作了*collections.abc.Mapping* 介面。

run(*callable*, **args*, ***kwargs*)

進入 Context，執行 `callable(*args, **kwargs)`，然後退出 Context。回傳 *callable* 的回傳值，如果發生例外（exception），則傳播例外。

例如：

```
import contextvars

var = contextvars.ContextVar('var')
var.set('spam')
print(var.get())  # 'spam'

ctx = contextvars.copy_context()

def main():
    # 'var' 之前被設成 'spam'
    # 呼叫 'copy_context()' 和 'ctx.run(main)', 所以:
    print(var.get())  # 'spam'
    print(ctx[var])  # 'spam'

    var.set('ham')

    # 現在, 在把 'var' 的值設成 'ham' 後:
    print(var.get())  # 'ham'
    print(ctx[var])  # 'ham'

# 'main' 函式對 'var' 所做的任何變更都會
# 包含在 'ctx' 🄵: .
ctx.run(main)

# 'main()' 函式是在 'ctx' 情境中執行,
# 所以對 'var' 的變更會保存在 'ctx' 中:
print(ctx[var])  # 'ham'

# 但是, 在 'ctx' 外, 'var' 的值仍然是 'spam':
print(var.get())  # 'spam'
```

copy()

回傳情境物件的淺層🄵本（shallow copy）。

var in context

如果情境🄵面有 *var* 的值，則回傳 `True`，否則回傳 `False`。

context[var]

回傳 *var* *ContextVar* 變數的值。如果該變數🄵🄵有在情境物件中設定，則會引發*KeyError* 錯誤。

get(*var*[, *default*])

如果 *var* 的值在情境物件中，則回傳 *var* 的值。否則回傳 *default*。如果🄵有 *default* 值，則回傳 `None`。

iter(context)

回傳儲存於情境物件中變數的🄵代器。

len(proxy)

回傳情境物件中的變數個數。

keys()

回傳情境物件中所有變數的串列。

values()

回傳情境物件中所有變數的值的串列。

items()

回傳情境物件中所有變數與其值的 2-元組（2-tuples）的串列。

18.10.3 對 asyncio 的支援

asyncio 原生支援情境變數，不需任何額外設定。舉例來🄵，以下是一個簡單的 echo 伺服器，使用情境變數讓遠端用🄵端的位址在處理該用🄵端的任務中可用：

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # 即使不把目前處理中的用🄵端 (client) 傳入此函式
    # 仍可取得其位址

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\r\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # 在任何我們呼叫的程式碼中，都可以
    # 呼叫 'client_addr_var.get()' 來取得用🄵端的位址

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break

    writer.write(b'HTTP/1.1 200 OK\r\n')  # status line
    writer.write(b'\r\n')  # headers
    writer.write(render_goodbye())  # body
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# 你可以使用 telnet 或 curl 測試:
#     telnet 127.0.0.1 8081
#     curl 127.0.0.1:8081
```

以下是支援部份上述服務的模組：

18.11 _thread --- 低階執行緒 API

這個模組提供了與多個執行緒（也稱🄵:dfn: *輕量級行程*或:dfn: *任務*）一起工作的低階原始物件 --- 多個控制執行緒分享其全域資料空間。🄵了處理同步問題，也提供了簡單的鎖 (lock) 機制（也稱🄵:dfn: *互斥鎖*或 *二進位號*🄵）。*threading* 模組提供了一個建立在這個模組之上的更易於使用和高階的執行緒 API。

在 3.7 版的變更: 這個模組之前是可選擇性的，但現在已經是可用的。

這個模組定義了以下的常數和函式：

exception _thread.**error**

在執行緒相關的錯誤發生時引發。

在 3.3 版的變更: 現在是🄵建例外*RuntimeError* 的🄵名。

_thread.**LockType**

這是鎖物件的型🄵。

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])

開始一個新的執行緒🄵回傳其識🄵字 (identifier) 。該執行緒執行帶有引數列表 *args**（必須是一個 *tuple*（元組））的函式 **function*。可選的 *kwargs* 引數指定一個關鍵字引數的字典。

當函式回傳時，執行緒會🄵默退出。

當函式因未處理的例外終止時，將呼叫*sys.unraisablehook()* 來處理該例外。🄵子引數的 *object* 屬性是 *function*。預設情🄵下，會列印堆🄵跟🄵，然後執行緒退出（但其他執行緒會繼續運行）。

當函式引發*SystemExit* 例外時，它會被🄵默忽略。

引發一個稽核事件 _thread.start_new_thread，帶有引數 function、args 和 kwargs。

在 3.8 版的變更: 現在使用*sys.unraisablehook()* 來處理未處理的例外。

_thread.**interrupt_main**(*signum=signal.SIGINT*, */*)

模擬一個訊號到達主執行緒的效果。執行緒可以使用此函式來中斷主執行緒，但無法保證中斷會立即發生。

如果提供了 *signum*，則模擬指定的訊號編號。如果未提供 *signum*，則模擬*signal.SIGINT* 訊號。

如果給定的訊號在 Python 中未被處理（即設置🄵*signal.SIG_DFL* 或*signal.SIG_IGN*），此函式不做任何操作。

在 3.10 版的變更: 新增了 *signum* 引數以自定義訊號編號。

> **備🄵**
>
> 這🄵不會發出對應的訊號，而是安排呼叫相應的處理器（如果存在的話）。如果你想真正發出訊號，請使用*signal.raise_signal()*。

_thread.**exit**()

引發*SystemExit* 例外。當未捕獲時，將導致執行緒🄵默退出。

_thread.**allocate_lock**()

回傳一個新的鎖物件。鎖物件的方法如下所述。初始狀🄵下鎖是解鎖狀態。

_thread.**get_ident**()

回傳目前執行緒的「執行緒識🄵字」。這是一個非零的整數。它的值🄵有直接的含義；它被用作一個 magic cookie，例如用於索引特定於執行緒的資料的字典。當執行緒退出🄵建立另一個執行緒時，執行緒識🄵字可能會被重🄵使用。

_thread.**get_native_id**()

回傳由核心 (kernel) 分配的目前執行緒的原生整數執行緒 ID。這是一個非負整數。它的值可用於在整個系統中唯一標識此特定執行緒（直到執行緒終止後，該值可能被操作系統重新使用）。

可用性: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

在 3.8 版被加入.

在 3.13 版的變更: 新增了對 GNU/kFreeBSD 的支援。

_thread.**stack_size**([*size*])

回傳在建立新執行緒時使用的執行緒堆🄵大小。可選的 *size* 引數指定了隨後建立的執行緒要使用的堆🄵大小，必須是 0（使用平台或配置的預設值）或至少 32,768（32 KiB）的正整數值。如果未指定 *size*，則使用 0。如果不支持更改執行緒堆🄵大小，則會引發*RuntimeError* 錯誤。如果指定的堆

🄵大小無效，則會引發*ValueError* 錯誤，🄵且堆🄵大小不會被修改。目前，32 KiB 是保證直譯器本身具有足🄵堆🄵空間所支持的最小堆🄵大小值。請注意，某些平台對於堆🄵大小的值可能有特定的限制，例如要求最小堆🄵大小 > 32 KiB，或要求按系統記憶體頁面大小的倍數進行分配。應參考平台文件以取得更多訊息（4 KiB 頁面是比較普遍的；在缺乏更具體訊息的情🄵下，建議使用 4096 的倍數作🄵堆🄵大小）。

可用性: Windows, pthreads.

Unix 平台上支援 POSIX 執行緒。

_thread.**TIMEOUT_MAX**

Lock.acquire 的 *timeout* 參數所允許的最大值。指定超過此值的 timeout 將引發*OverflowError* 錯誤。

在 3.2 版被加入.

鎖物件具有以下方法：

lock.**acquire**(*blocking=True*, *timeout=-1*)

🄵有任何可選引數時，此方法無條件地取得鎖，必要時會等待直到被另一個執行緒釋放（一次只能有一個執行緒取得鎖 --- 這正是鎖存在的原因）。

如果存在 *blocking* 引數，則根據其值執行操作：如果🄵 False，只有在可以立即獲取鎖而無需等待的情🄵下才取得鎖，而如果🄵 True，則像上面一樣無條件地取得鎖。

如果存在浮點數的 *timeout* 引數且🄵正值，則它指定了在回傳之前的最大等待時間（以秒🄵單位）。如果 *timeout* 引數🄵負值，則表示等待時間會無限期地等待。如果 *blocking* 🄵 False，則你無法指定 *timeout*。

如果成功取得鎖，回傳值🄵 `True`，否則🄵 `False`。

在 3.2 版的變更: 新增的 *timeout* 參數。

在 3.2 版的變更: 現在取得鎖的操作可以被 POSIX 訊號中斷。

在 3.14 版的變更: Lock acquires can now be interrupted by signals on Windows.

lock.**release**()

釋放鎖。鎖必須先前被取得，但不一定是由同一個執行緒取得的。

lock.**locked**()

回傳鎖的狀態：如果鎖已被某個執行緒取得，則回傳 `True`，否則回傳 `False`。

除了這些方法之外，鎖物件還可以透過 `with` 語句來使用，例如：

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock 在執行這🄵時被鎖定")
```

注意事項：

- 中斷總會跳到主執行緒（該執行緒將接收*KeyboardInterrupt* 例外。）
- 呼叫*sys.exit()* 函式或引發*SystemExit* 例外等同於呼叫*_thread.exit()* 函式。
- 當主執行緒退出時，其他執行緒是否保留是由系統🄵定的。在大多數系統上，它們將被終止，而不會執行 `try` ... `finally` 子句或執行物件的解構函式。

CHAPTER 19

網路與行程間通訊

本章所描述的模組提供了網路和行程間通訊的機制。

某些模組僅適用於同一台機器上的兩個行程，例如*signal* 和*mmap*。其他模組則支援網路通訊協定，可供兩個或多個行程跨機器進行通訊。

本章所述的模組列表如下：

19.1 asyncio --- 非同步 I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio 是讓使用者以 **async/await** 語法來編寫🄵行 *(concurrent)* 程式碼的函式庫 (library)。

asyncio 作🄵多個 Python 非同步框架的基礎，在高效能網路與網頁伺服器、資料庫連🄵函式庫、分散式任務🄵列等服務都可以看得到它。

asyncio 往往是個建構 IO 密集型與高階層**結構化**網路程式碼的完美選擇。

也參考

a-conceptual-overview-of-asyncio
asyncio 的基本原理🄵明。

asyncio 提供了一系列**高階** API：

- 🄵行地運行 *Python* 協程 *(coroutine)* 🄵擁有完整控制權；

- 執行網路 *IO* 與 *IPC*；
- 控制子行程 *(subprocess)*；
- 透過🄵列 *(queue)* 分配任務；
- 同步🄵行程式碼；

此外，還有一些給函式庫與框架 *(framework)* 開發者的**低階** API：

- 建立與管理*event loops*（事件🄵圈），它提供了能被用於網路、執行子行程、處理作業系統訊號等任務的非同步 API；
- 使用*transports*（*asyncio* 底層傳輸相關類🄵）來實作高效能協定；
- 透過 async/await 語法來橋接基於回呼 (callback-based) 的函式庫與程式碼。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

asyncio REPL

你能在*REPL* 中對一個 `asyncio` 的🄵行情境 (context) 進行實驗：

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

🄵生一個稽核事件 `cpython.run_stdin` 且🄵有引數。

在 3.12.5 版的變更: (也包括 3.11.10、3.10.15、3.9.20 及 3.8.20）發出稽核事件。

在 3.13 版的變更: 可能的話使用 PyREPL，在這種情🄵下 `PYTHONSTARTUP` 也會被執行。發出稽核事件。

參🄵

19.1.1 Runners (執行器)

原始碼： Lib/asyncio/runners.py

這個章節概述用於執行 asyncio 程式碼的高階 asyncio 原始物件。

他們是基於一個事件🄵圈，目的是🄵了簡化常見且廣泛運用場景的非同步程式碼。

- *運行一個 asyncio 程式*
- *Runner context manager*
- *Handling Keyboard Interruption*

運行一個 asyncio 程式

`asyncio.`**`run`**(*coro*, *, *debug=None*, *loop_factory=None*)

在 asyncio 事件🄵圈中執行 *coro* 🄵回傳結果。

該引數可以是任何可等待物件 (awaitable object)。

這個函式負責運行被傳入的可等待物件、管理 asyncio 的事件🄵圈、終結非同步🄵生器以及關閉執行器。

當另一個非同步事件🄵圈在同一執行緒中執行時，無法呼叫此函式。

如果 *debug* 🄵 `True`，事件🄵圈會以除錯模式執行。`False` 則會關閉除錯模式。`None` 則會優先使用除錯模式的全域設定。

如果 *loop_factory* 不🄵 `None`，它會被用於建立一個新的事件🄵圈；否則會改用 *asyncio.new_event_loop()*。🄵圈會在最後關閉。這個函式應該要作🄵 asyncio 程式的主要進入點，且理想上僅會被呼叫一次。推薦使用 *loop_factory* 來設定事件🄵圈時而不是使用 policies（政策）。傳遞 *asyncio.EventLoop* 可以讓 asyncio 在🄵有政策系統的情🄵下運行。

The executor is given a timeout duration of 5 minutes to shutdown. If the executor hasn't finished within that duration, a warning is emitted and the executor is closed.

範例：

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

在 3.7 版被加入.

在 3.9 版的變更: Updated to use *loop.shutdown_default_executor()*.

在 3.10 版的變更: *debug* is `None` by default to respect the global debug mode settings.

在 3.12 版的變更: 新增 *loop_factory* 參數。

在 3.14 版的變更: *coro* can be any awaitable object.

備🄵

The `asyncio` policy system is deprecated and will be removed in Python 3.16; from there on, an explicit *loop_factory* is needed to configure the event loop.

Runner context manager

class asyncio.**Runner**(*, *debug=None*, *loop_factory=None*)

A context manager that simplifies *multiple* async function calls in the same context.

Sometimes several top-level async functions should be called in the same *event loop* and *contextvars.Context*.

如果 *debug* 🄵 `True`，事件🄵圈會以除錯模式執行。`False` 則會關閉除錯模式。`None` 則會優先使用除錯模式的全域設定。

loop_factory could be used for overriding the loop creation. It is the responsibility of the *loop_factory* to set the created loop as the current one. By default *asyncio.new_event_loop()* is used and set as current event loop with *asyncio.set_event_loop()* if *loop_factory* is `None`.

Basically, *asyncio.run()* example can be rewritten with the runner usage:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

在 3.11 版被加入.

run(*coro*, *, *context=None*)

Execute *coro* in the embedded event loop.

該引數可以是任何可等待物件 (awaitable object)。

If the argument is a coroutine, it is wrapped in a Task.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the code to run in. The runner's default context is used if context is `None`.

Returns the awaitable's result or raises an exception.

當另一個非同步事件🄵圈在同一執行緒中執行時，無法呼叫此函式。

在 3.14 版的變更: *coro* can be any awaitable object.

close()

Close the runner.

Finalize asynchronous generators, shutdown default executor, close the event loop and release embedded *contextvars.Context*.

get_loop()

Return the event loop associated with the runner instance.

備🄵

Runner uses the lazy initialization strategy, its constructor doesn't initialize underlying low-level structures.

Embedded *loop* and *context* are created at the `with` body entering or the first call of *run()* or *get_loop()*.

Handling Keyboard Interruption

在 3.11 版被加入.

When *signal.SIGINT* is raised by `Ctrl-C`, *KeyboardInterrupt* exception is raised in the main thread by default. However this doesn't work with *asyncio* because it can interrupt asyncio internals and can hang the program from exiting.

To mitigate this issue, *asyncio* handles *signal.SIGINT* as follows:

1. *asyncio.Runner.run()* installs a custom *signal.SIGINT* handler before any user code is executed and removes it when exiting from the function.
2. The *Runner* creates the main task for the passed coroutine for its execution.
3. When *signal.SIGINT* is raised by `Ctrl-C`, the custom signal handler cancels the main task by calling *asyncio.Task.cancel()* which raises *asyncio.CancelledError* inside the main task. This causes the Python stack to unwind, `try/except` and `try/finally` blocks can be used for resource cleanup. After the main task is cancelled, *asyncio.Runner.run()* raises *KeyboardInterrupt*.
4. A user could write a tight loop which cannot be interrupted by *asyncio.Task.cancel()*, in which case the second following `Ctrl-C` immediately raises the *KeyboardInterrupt* without cancelling the main task.

19.1.2 協程與任務

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

- 協程
- *Awaitables*
- *Creating Tasks*
- *Task Cancellation*
- *Task Groups*

- *Sleeping*
- *Running Tasks Concurrently*
- *Eager Task Factory*
- *Shielding From Cancellation*
- *Timeouts*
- *Waiting Primitives*
- *Running in Threads*
- *Scheduling From Other Threads*
- *Introspection*
- *Task 物件*

協程

原始碼： Lib/asyncio/coroutines.py

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints ”hello”, waits 1 second, and then prints ”world”:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine, asyncio provides the following mechanisms:

- The `asyncio.run()` function to run the top-level entry point ”main()” function (see the above example.)
- Awaiting on a coroutine. The following snippet of code will print ”hello” after waiting for 1 second, and then print ”world” after waiting for *another* 2 seconds:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')
```

(繼續下一頁)

(繼續上一頁)

```
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

預期的輸出:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- The *asyncio.create_task()* function to run coroutines concurrently as asyncio *Tasks*.

  Let's modify the above example and run two `say_after` coroutines *concurrently*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

  Note that expected output now shows that the snippet runs 1 second faster than before:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

- The *asyncio.TaskGroup* class provides a more modern alternative to *create_task()*. Using this API, the last example becomes:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(
            say_after(1, 'hello'))

        task2 = tg.create_task(
            say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")

    # The await is implicit when the context manager exits.

    print(f"finished at {time.strftime('%X')}")
```

  The timing and output should be the same as for the previous version.

  在 3.11 版被加入: *asyncio.TaskGroup*。

Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of *awaitable* objects: **coroutines**, **Tasks**, and **Futures**.

協程

Python coroutines are *awaitables* and therefore can be awaited from other coroutines:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()  # will raise a "RuntimeWarning".

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

重要

In this documentation the term ”coroutine” can be used for two closely related concepts:

- a *coroutine function*: an `async def` function;
- a *coroutine object*: an object returned by calling a *coroutine function*.

Tasks

Tasks are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like *`asyncio.create_task()`* the coroutine is automatically scheduled to run soon:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A *Future* is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.

When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is *loop.run_in_executor()*.

Creating Tasks

原始碼： Lib/asyncio/tasks.py

---

asyncio.**create_task**(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

Wrap the *coro* *coroutine* into a *Task* and schedule its execution. Return the Task object.

The full function signature is largely the same as that of the *Task* constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. The current context copy is created when no *context* is provided.

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by *loop.set_task_factory()* will be used.

The task is executed in the loop returned by *get_running_loop()*, *RuntimeError* is raised if there is no running loop in current thread.

> **備🄵**
>
> *asyncio.TaskGroup.create_task()* is a new alternative leveraging structural concurrency; it allows for waiting for a group of related tasks with strong safety guarantees.

> **重要**
>
> Save a reference to the result of this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done. For reliable "fire-and-forget" background tasks, gather them in a collection:
>
> ```
> background_tasks = set()
>
> for i in range(10):
>     task = asyncio.create_task(some_coro(param=i))
>
>     # Add task to the set. This creates a strong reference.
>     background_tasks.add(task)
> ```

```
    # To prevent keeping references to finished tasks forever,
    # make each task remove its own reference from the set after
    # completion:
    task.add_done_callback(background_tasks.discard)
```

在 3.7 版被加入.

在 3.8 版的變更: 新增 *name* 參數。

在 3.11 版的變更: 新增 *context* 參數。

在 3.14 版的變更: Added the *eager_start* parameter by passing on all *kwargs*.

Task Cancellation

Tasks can easily and safely be cancelled. When a task is cancelled, *asyncio.CancelledError* will be raised in the task at the next opportunity.

It is recommended that coroutines use `try/finally` blocks to robustly perform clean-up logic. In case *asyncio.CancelledError* is explicitly caught, it should generally be propagated when clean-up is complete. *asyncio.CancelledError* directly subclasses *BaseException* so most code will not need to be aware of it.

The asyncio components that enable structured concurrency, like *asyncio.TaskGroup* and *asyncio.timeout()*, are implemented using cancellation internally and might misbehave if a coroutine swallows *asyncio.CancelledError*. Similarly, user code should not generally call *uncancel*. However, in cases when suppressing *asyncio.CancelledError* is truly desired, it is necessary to also call `uncancel()` to completely remove the cancellation state.

Task Groups

Task groups combine a task creation API with a convenient and reliable way to wait for all tasks in the group to finish.

class asyncio.**TaskGroup**

An asynchronous context manager holding a group of tasks. Tasks can be added to the group using *create_task()*. All tasks are awaited when the context manager exits.

在 3.11 版被加入.

create_task(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

Create a task in this task group. The signature matches that of *asyncio.create_task()*. If the task group is inactive (e.g. not yet entered, already finished, or in the process of shutting down), we will close the given `coro`.

在 3.13 版的變更: Close the given coroutine if the task group is not active.

在 3.14 版的變更: Passes on all *kwargs* to *loop.create_task()*

範例:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()}, {task2.result()}")
```

The `async with` statement will wait for all tasks in the group to finish. While waiting, new tasks may still be added to the group (for example, by passing `tg` into one of the coroutines and calling `tg.create_task()` in that coroutine). Once the last task has finished and the `async with` block is exited, no new tasks may be added to the group.

The first time any of the tasks belonging to the group fails with an exception other than *asyncio.CancelledError*, the remaining tasks in the group are cancelled. No further tasks can then be added to the group. At this point, if

the body of the `async with` statement is still active (i.e., `__aexit__()` hasn't been called yet), the task directly containing the `async with` statement is also cancelled. The resulting *asyncio.CancelledError* will interrupt an `await`, but it will not bubble out of the containing `async with` statement.

Once all tasks have finished, if any tasks have failed with an exception other than *asyncio.CancelledError*, those exceptions are combined in an *ExceptionGroup* or *BaseExceptionGroup* (as appropriate; see their documentation) which is then raised.

Two base exceptions are treated specially: If any task fails with *KeyboardInterrupt* or *SystemExit*, the task group still cancels the remaining tasks and waits for them, but then the initial *KeyboardInterrupt* or *SystemExit* is re-raised instead of *ExceptionGroup* or *BaseExceptionGroup*.

If the body of the `async with` statement exits with an exception (so `__aexit__()` is called with an exception set), this is treated the same as if one of the tasks failed: the remaining tasks are cancelled and then waited for, and non-cancellation exceptions are grouped into an exception group and raised. The exception passed into `__aexit__()`, unless it is *asyncio.CancelledError*, is also included in the exception group. The same special case is made for *KeyboardInterrupt* and *SystemExit* as in the previous paragraph.

Task groups are careful not to mix up the internal cancellation used to "wake up" their `__aexit__()` with cancellation requests for the task in which they are running made by other parties. In particular, when one task group is syntactically nested in another, and both experience an exception in one of their child tasks simultaneously, the inner task group will process its exceptions, and then the outer task group will receive another cancellation and process its own exceptions.

In the case where a task group is cancelled externally and also must raise an *ExceptionGroup*, it will call the parent task's *cancel()* method. This ensures that a *asyncio.CancelledError* will be raised at the next `await`, so the cancellation is not lost.

Task groups preserve the cancellation count reported by *asyncio.Task.cancelling()*.

在 3.13 版的變更: Improved handling of simultaneous internal and external cancellations and correct preservation of cancellation counts.

Terminating a Task Group

While terminating a task group is not natively supported by the standard library, termination can be achieved by adding an exception-raising task to the task group and ignoring the raised exception:

```
import asyncio
from asyncio import TaskGroup

class TerminateTaskGroup(Exception):
    """Exception raised to terminate a task group."""

async def force_terminate_task_group():
    """Used to force termination of a task group."""
    raise TerminateTaskGroup()

async def job(task_id, sleep_time):
    print(f'Task {task_id}: start')
    await asyncio.sleep(sleep_time)
    print(f'Task {task_id}: done')

async def main():
    try:
        async with TaskGroup() as group:
            # spawn some tasks
            group.create_task(job(1, 0.5))
            group.create_task(job(2, 1.5))
            # sleep for 1 second
            await asyncio.sleep(1)
            # add an exception-raising task to force the group to terminate
            group.create_task(force_terminate_task_group())
```

(繼續下一頁)

(繼續上一頁)

```
    except* TerminateTaskGroup:
        pass

asyncio.run(main())
```

預期的輸出：

```
Task 1: start
Task 2: start
Task 1: done
```

Sleeping

async asyncio.**sleep**(*delay*, *result=None*)

Block for *delay* seconds.

If *result* is provided, it is returned to the caller when the coroutine completes.

`sleep()` always suspends the current task, allowing other tasks to run.

Setting the delay to 0 provides an optimized path to allow other tasks to run. This can be used by long-running functions to avoid blocking the event loop for the full duration of the function call.

Example of coroutine displaying the current date every second for 5 seconds:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

在 3.10 版的變更: 移除 *loop* 參數。

在 3.13 版的變更: 如果 *delay* 是*nan*，則引發*ValueError*。

Running Tasks Concurrently

awaitable asyncio.**gather**(**aws*, *return_exceptions=False*)

Run *awaitable objects* in the *aws* sequence *concurrently*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in *aws*.

If *return_exceptions* is `False` (default), the first raised exception is immediately propagated to the task that awaits on `gather()`. Other awaitables in the *aws* sequence **won't be cancelled** and will continue to run.

If *return_exceptions* is `True`, exceptions are treated the same as successful results, and aggregated in the result list.

If `gather()` is *cancelled*, all submitted awaitables (that have not completed yet) are also *cancelled*.

If any Task or Future from the *aws* sequence is *cancelled*, it is treated as if it raised *CancelledError* -- the `gather()` call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

> **備🄵**
>
> A new alternative to create and run tasks concurrently and wait for their completion is `asyncio.TaskGroup`. *TaskGroup* provides stronger safety guarantees than *gather* for scheduling a nesting of subtasks: if a task (or a subtask, a task scheduled by a task) raises an exception, *TaskGroup* will, while *gather* will not, cancel the remaining scheduled tasks).

範例：

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

> **備🄵**
>
> If *return_exceptions* is false, cancelling gather() after it has been marked done won't cancel any submitted awaitables. For instance, gather can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching an exception (raised by one of the awaitables) from gather won't cancel any other awaitables.

在 3.7 版的變更: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

在 3.10 版的變更: 移除 *loop* 參數。

在 3.10 版之後被🄴用: Deprecation warning is emitted if no positional arguments are provided or not all positional arguments are Future-like objects and there is no running event loop.

Eager Task Factory

asyncio.**eager_task_factory**(*loop*, *coro*, *, *name=None*, *context=None*)

A task factory for eager task execution.

When using this factory (via *loop.set_task_factory(asyncio.eager_task_factory)*), coroutines begin execution synchronously during *Task* construction. Tasks are only scheduled on the event loop if they block. This can be a performance improvement as the overhead of loop scheduling is avoided for coroutines that complete synchronously.

A common example where this is beneficial is coroutines which employ caching or memoization to avoid actual I/O when possible.

> **備🄵**
>
> Immediate execution of the coroutine is a semantic change. If the coroutine returns or raises, the task is never scheduled to the event loop. If the coroutine execution blocks, the task is scheduled to the event loop. This change may introduce behavior changes to existing applications. For example, the application's task execution order is likely to change.

在 3.12 版被加入.

asyncio.**create_eager_task_factory**(*custom_task_constructor*)

Create an eager task factory, similar to *eager_task_factory()*, using the provided *custom_task_constructor* when creating a new task instead of the default *Task*.

custom_task_constructor must be a *callable* with the signature matching the signature of *Task.__init__*. The callable must return a *asyncio.Task*-compatible object.

This function returns a *callable* intended to be used as a task factory of an event loop via *loop.set_task_factory(factory)*).

在 3.12 版被加入.

Shielding From Cancellation

awaitable asyncio.**shield**(*aw*)

Protect an *awaitable object* from being *cancelled*.

If *aw* is a coroutine it is automatically scheduled as a Task.

The statement:

```
task = asyncio.create_task(something())
res = await shield(task)
```

is equivalent to:

```
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the "await" expression still raises a *CancelledError*.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a try/except clause, as follows:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
```

(繼續下一頁)

(繼續上一頁)

```
except CancelledError:
    res = None
```

重要

Save a reference to tasks passed to this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn't referenced elsewhere may get garbage collected at any time, even before it's done.

在 3.10 版的變更: 移除 *loop* 參數。

在 3.10 版之後被🄵用: Deprecation warning is emitted if *aw* is not Future-like object and there is no running event loop.

Timeouts

`asyncio.timeout`(*delay*)

Return an asynchronous context manager that can be used to limit the amount of time spent waiting on something.

delay can either be `None`, or a float/int number of seconds to wait. If *delay* is `None`, no time limit will be applied; this can be useful if the delay is unknown when the context manager is created.

In either case, the context manager can be rescheduled after creation using *Timeout.reschedule()*.

範例:

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

If `long_running_task` takes more than 10 seconds to complete, the context manager will cancel the current task and handle the resulting *asyncio.CancelledError* internally, transforming it into a *TimeoutError* which can be caught and handled.

備🄵

The *asyncio.timeout()* context manager is what transforms the *asyncio.CancelledError* into a *TimeoutError*, which means the *TimeoutError* can only be caught *outside* of the context manager.

Example of catching *TimeoutError*:

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

The context manager produced by *asyncio.timeout()* can be rescheduled to a different deadline and inspected.

class `asyncio.Timeout`(*when*)

An asynchronous context manager for cancelling overdue coroutines.

`when` should be an absolute time at which the context should time out, as measured by the event loop's clock:

- If `when` is `None`, the timeout will never trigger.
- If `when < loop.time()`, the timeout will trigger on the next iteration of the event loop.

when() → *float* | *None*
Return the current deadline, or `None` if the current deadline is not set.

reschedule(*when:* float | None)
Reschedule the timeout.

expired() → *bool*
Return whether the context manager has exceeded its deadline (expired).

範例：

```
async def main():
    try:
        # We do not know the timeout when starting, so we pass ``None``.
        async with asyncio.timeout(None) as cm:
            # We know the timeout now, so we reschedule it.
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

Timeout context managers can be safely nested.

在 3.11 版被加入.

asyncio.**timeout_at**(*when*)
Similar to *asyncio.timeout()*, except *when* is the absolute time to stop waiting, or `None`.

範例：

```
async def main():
    loop = get_running_loop()
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

在 3.11 版被加入.

async asyncio.**wait_for**(*aw*, *timeout*)
Wait for the *aw* *awaitable* to complete with a timeout.

If *aw* is a coroutine it is automatically scheduled as a Task.

timeout can either be `None` or a float or int number of seconds to wait for. If *timeout* is `None`, block until the future completes.

If a timeout occurs, it cancels the task and raises *TimeoutError*.

To avoid the task *cancellation*, wrap it in *shield()*.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*. If an exception happens during cancellation, it is propagated.

If the wait is cancelled, the future *aw* is also cancelled.

範例：

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

在 3.7 版的變更: When *aw* is cancelled due to a timeout, `wait_for` waits for *aw* to be cancelled. Previously, it raised *TimeoutError* immediately.

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: 引發*TimeoutError* 而不是*asyncio.TimeoutError*。

Waiting Primitives

async asyncio.**wait**(*aws*, *, *timeout=None*, *return_when=ALL_COMPLETED*)

Run *Future* and *Task* instances in the *aws* iterable concurrently and block until the condition specified by *return_when*.

The *aws* iterable must not be empty.

Returns two sets of Tasks/Futures: `(done, pending)`.

用法：

```
done, pending = await asyncio.wait(aws)
```

timeout (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise *TimeoutError*. Futures or Tasks that aren't done when the timeout occurs are simply returned in the second set.

return_when indicates when this function should return. It must be one of the following constants:

常數	描述
asyncio.**FIRST_COMPLETED**	The function will return when any future finishes or is cancelled.
asyncio.**FIRST_EXCEPTION**	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to *ALL_COMPLETED*.
asyncio.**ALL_COMPLETED**	The function will return when all futures finish or are cancelled.

Unlike *wait_for()*, `wait()` does not cancel the futures when a timeout occurs.

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: Passing coroutine objects to `wait()` directly is forbidden.

在 3.12 版的變更: Added support for generators yielding tasks.

`asyncio.`**`as_completed`**(*aws*, *, *timeout=None*)

Run *awaitable objects* in the *aws* iterable concurrently. The returned object can be iterated to obtain the results of the awaitables as they finish.

The object returned by `as_completed()` can be iterated as an *asynchronous iterator* or a plain *iterator*. When asynchronous iteration is used, the originally-supplied awaitables are yielded if they are tasks or futures. This makes it easy to correlate previously-scheduled tasks with their results. Example:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect is done. The result can be obtained by
    # awaiting it or calling earliest_connect.result()
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

During asynchronous iteration, implicitly-created tasks will be yielded for supplied awaitables that aren't tasks or futures.

When used as a plain iterator, each iteration yields a new coroutine that returns the result or raises the exception of the next completed awaitable. This pattern is compatible with Python versions older than 3.13:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect is not one of the original task objects. It must be
    # awaited to obtain the result value or raise the exception of the
    # awaitable that finishes next.
    reader, writer = await next_connect
```

A *TimeoutError* is raised if the timeout occurs before all awaitables are done. This is raised by the `async for` loop during asynchronous iteration or by the coroutines yielded during plain iteration.

在 3.10 版的變更: 移除 *loop* 參數。

在 3.10 版之後被🄴用: Deprecation warning is emitted if not all awaitable objects in the *aws* iterable are Future-like objects and there is no running event loop.

在 3.12 版的變更: Added support for generators yielding tasks.

在 3.13 版的變更: The result can now be used as either an *asynchronous iterator* or as a plain *iterator* (previously it was only a plain iterator).

Running in Threads

`async` `asyncio.`**`to_thread`**(*func*, */*, **args*, ***kwargs*)

Asynchronously run function *func* in a separate thread.

Any *args and **kwargs supplied for this function are directly passed to *func*. Also, the current *contextvars.Context* is propagated, allowing context variables from the event loop thread to be accessed in the separate thread.

Return a coroutine that can be awaited to get the eventual result of *func*.

This coroutine function is primarily intended to be used for executing IO-bound functions/methods that would otherwise block the event loop if they were run in the main thread. For example:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Directly calling `blocking_io()` in any coroutine would block the event loop for its duration, resulting in an additional 1 second of run time. Instead, by using `asyncio.to_thread()`, we can run it in a separate thread without blocking the event loop.

備F

Due to the *GIL*, `asyncio.to_thread()` can typically only be used to make IO-bound functions non-blocking. However, for extension modules that release the GIL or alternative Python implementations that don't have one, `asyncio.to_thread()` can also be used for CPU-bound functions.

在 3.9 版被加入.

Scheduling From Other Threads

`asyncio.`**`run_coroutine_threadsafe`**(*coro*, *loop*)

Submit a coroutine to the given event loop. Thread-safe.

Return a *`concurrent.futures.Future`* to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```
def in_thread(loop: asyncio.AbstractEventLoop) -> None:
    # Run some blocking IO
    pathlib.Path("example.txt").write_text("hello world", encoding="utf8")

    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
```

(繼續下一頁)

(繼續上一頁)

```
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3

async def amain() -> None:
    # Get the running loop
    loop = asyncio.get_running_loop()

    # Run something in a thread
    await asyncio.to_thread(in_thread, loop)
```

It's also possible to run the other way around. Example:

```
@contextlib.contextmanager
def loop_in_thread() -> Generator[asyncio.AbstractEventLoop]:
    loop_fut = concurrent.futures.Future[asyncio.AbstractEventLoop]()
    stop_event = asyncio.Event()

    async def main() -> None:
        loop_fut.set_result(asyncio.get_running_loop())
        await stop_event.wait()

    with concurrent.futures.ThreadPoolExecutor(1) as tpe:
        complete_fut = tpe.submit(asyncio.run, main())
        for fut in concurrent.futures.as_completed((loop_fut, complete_fut)):
            if fut is loop_fut:
                loop = loop_fut.result()
                try:
                    yield loop
                finally:
                    loop.call_soon_threadsafe(stop_event.set)
            else:
                fut.result()

# Create a loop in another thread
with loop_in_thread() as loop:
    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

See the *concurrency and multithreading* section of the documentation.

Unlike other asyncio functions this function requires the *loop* argument to be passed explicitly.

在 3.5.1 版被加入.

Introspection

asyncio.**current_task**(*loop=None*)

Return the currently running *Task* instance, or `None` if no task is running.

If *loop* is `None` *get_running_loop()* is used to get the current loop.

在 3.7 版被加入.

asyncio.**all_tasks**(*loop=None*)

Return a set of not yet finished *Task* objects run by the loop.

If *loop* is `None`, *get_running_loop()* is used for getting current loop.

在 3.7 版被加入.

asyncio.**iscoroutine**(*obj*)

Return `True` if *obj* is a coroutine object.

在 3.4 版被加入.

Task 物件

class asyncio.**Task**(*coro*, *, *loop=None*, *name=None*, *context=None*, *eager_start=False*)

A *Future-like* object that runs a Python *coroutine*. Not thread-safe.

Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level *asyncio.create_task()* function to create Tasks, or the low-level *loop.create_task()* or *ensure_future()* functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the *cancel()* method. Calling it will cause the Task to throw a *CancelledError* exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

cancelled() can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the *CancelledError* exception and was actually cancelled.

asyncio.Task inherits from *Future* all of its APIs except *Future.set_result()* and *Future.set_exception()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *coro* to run in. If no *context* is provided, the Task copies the current context and later runs its coroutine in the copied context.

An optional keyword-only *eager_start* argument allows eagerly starting the execution of the *asyncio.Task* at task creation time. If set to `True` and the event loop is running, the task will start executing the coroutine immediately, until the first time the coroutine blocks. If the coroutine returns or raises without blocking, the task will be finished eagerly and will skip scheduling to the event loop.

在 3.7 版的變更: 新增對*contextvars* 模組的支援。

在 3.8 版的變更: 新增 *name* 參數。

在 3.10 版之後被🄵用: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

在 3.11 版的變更: 新增 *context* 參數。

在 3.12 版的變更: 新增 *eager_start* 參數。

done()

Return `True` if the Task is *done*.

A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

result()

Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

If the Task has been *cancelled*, this method raises a *CancelledError* exception.

If the Task's result isn't yet available, this method raises an *InvalidStateError* exception.

exception()

Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

If the Task has been *cancelled*, this method raises a *CancelledError* exception.

If the Task isn't *done* yet, this method raises an *InvalidStateError* exception.

add_done_callback(*callback*, *, *context=None*)

Add a callback to be run when the Task is *done*.

This method should only be used in low-level callback-based code.

See the documentation of *Future.add_done_callback()* for more details.

remove_done_callback(*callback*)

Remove *callback* from the callbacks list.

This method should only be used in low-level callback-based code.

See the documentation of *Future.remove_done_callback()* for more details.

get_stack(*, *limit=None*)

Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional *limit* argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

print_stack(*, *limit=None*, *file=None*)

Print the stack or traceback for this Task.

This produces output similar to that of the traceback module for the frames retrieved by *get_stack()*.

The *limit* argument is passed to *get_stack()* directly.

The *file* argument is an I/O stream to which the output is written; by default output is written to *sys.stdout*.

get_coro()

Return the coroutine object wrapped by the *Task*.

> **備🄵**
>
> This will return None for Tasks which have already completed eagerly. See the *Eager Task Factory*.

在 3.8 版被加入.

在 3.12 版的變更: Newly added eager task execution means result may be None.

get_context()

Return the *contextvars.Context* object associated with the task.

在 3.12 版被加入.

get_name()

Return the name of the Task.

If no name has been explicitly assigned to the Task, the default asyncio Task implementation generates a default name during instantiation.

在 3.8 版被加入.

set_name(*value*)

Set the name of the Task.

The *value* argument can be any object, which is then converted to a string.

In the default Task implementation, the name will be visible in the *repr()* output of a task object.

在 3.8 版被加入.

cancel(*msg=None*)

Request the Task to be cancelled.

If the Task is already *done* or *cancelled*, return False, otherwise, return True.

The method arranges for a *CancelledError* exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a try ... ... except CancelledError ... finally block. Therefore, unlike *Future.cancel()*, *Task.cancel()* does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged. Should the coroutine nevertheless decide to suppress the cancellation, it needs to call *Task.uncancel()* in addition to catching the exception.

在 3.9 版的變更: 新增 *msg* 參數。

在 3.11 版的變更: The msg parameter is propagated from cancelled task to its awaiter. The following example illustrates how coroutines can intercept the cancellation request:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')
```

(繼續下一頁)

(繼續上一頁)

```
async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Return `True` if the Task is *cancelled*.

The Task is *cancelled* when the cancellation was requested with *cancel()* and the wrapped coroutine propagated the *CancelledError* exception thrown into it.

uncancel()

Decrement the count of cancellation requests to this Task.

Returns the remaining number of cancellation requests.

Note that once execution of a cancelled task completed, further calls to *uncancel()* are ineffective.

在 3.11 版被加入.

This method is used by asyncio's internals and isn't expected to be used by end-user code. In particular, if a Task gets successfully uncancelled, this allows for elements of structured concurrency like *Task Groups* and *asyncio.timeout()* to continue running, isolating cancellation to the respective structured block. For example:

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # Structured block affected by the timeout:
            await make_request()
            await make_another_request()
    except TimeoutError:
        log("There was a timeout")
    # Outer code not affected by the timeout:
    await unrelated_code()
```

While the block with `make_request()` and `make_another_request()` might get cancelled due to the timeout, `unrelated_code()` should continue running even in case of the timeout. This is implemented with *uncancel()*. *TaskGroup* context managers use *uncancel()* in a similar fashion.

If end-user code is, for some reason, suppressing cancellation by catching *CancelledError*, it needs to call this method to remove the cancellation state.

When this method decrements the cancellation count to zero, the method checks if a previous *cancel()* call had arranged for *CancelledError* to be thrown into the task. If it hasn't been thrown yet, that arrangement will be rescinded (by resetting the internal `_must_cancel` flag).

在 3.13 版的變更: Changed to rescind pending cancellation requests upon reaching zero.

cancelling()

Return the number of pending cancellation requests to this Task, i.e., the number of calls to *cancel()* less the number of *uncancel()* calls.

Note that if this number is greater than zero but the Task is still executing, *cancelled()* will still return `False`. This is because this number can be lowered by calling *uncancel()*, which can lead to the task not being cancelled after all if the cancellation requests go down to zero.

This method is used by asyncio's internals and isn't expected to be used by end-user code. See *uncancel()* for more details.

在 3.11 版被加入.

19.1.3 串流

原始碼： Lib/asyncio/streams.py

串流是支援 async/await (async/await-ready) 的高階原始物件 (high-level primitive)，用於處理網路連🄵。串流不需要使用回呼 (callback) 或低階協定和傳輸 (transport) 就能🄵傳送和接收資料。

這是一個使用 asyncio 串流編寫的 TCP echo 用🄵端範例：

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

另請參🄵下方*Examples* 段落。

串流函式

下面的高階 asyncio 函式可以用來建立和處理串流：

async asyncio.**open_connection**(*host=None*, *port=None*, ***, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

建立網路連🄵🄵回傳一對 `(reader, writer)` 物件。

回傳的 *reader* 和 *writer* 物件是*StreamReader* 和*StreamWriter* 類🄵的實例。

limit 指定了回傳的*StreamReader* 實例所使用的緩衝區 (buffer) 大小限制。*limit* 預設🄵 64 KiB。

其余的引數會直接傳遞到*loop.create_connection()*。

> **備🄵**
>
> The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 參數。

在 3.8 版的變更: 新增 *happy_eyeballs_delay* 和 *interleave* 參數。

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: 新增 *ssl_shutdown_timeout* 參數。

async asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, *, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

🄵動 socket 伺服器。

當一個新的用🄵端連🄵被建立時，回呼函式 *client_connected_cb* 就會被呼叫。該函式會接收到一對引數 `(reader, writer)`，分🄵🄵`StreamReader` 和`StreamWriter` 的實例。

client_connected_cb 既可以是普通的可呼叫物件 (callable)，也可以是一個協程函式；如果它是一個協程函式，它將自動作🄵`Task` 來被排程。

limit 指定了回傳的`StreamReader` 實例所使用的緩衝區 (buffer) 大小限制。*limit* 預設🄵 64 KiB。

剩下的引數將會直接傳遞給`loop.create_server()`。

> **備🄵**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 與 *start_serving* 參數。

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: 新增 *ssl_shutdown_timeout* 參數。

在 3.13 版的變更: 新增 *keep_alive* 參數。

Unix Sockets

async asyncio.**open_unix_connection**(*path=None*, *, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

建立一個 Unix socket 連🄵🄵回傳一對 `(reader, writer)`。

與`open_connection()` 相似，但是是操作 Unix sockets。

另請參🄵`loop.create_unix_connection()` 文件。

> **備🄵**
>
> The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

可用性: Unix.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 參數。*path* 參數現在可以是個*path-like object*

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: 新增 *ssl_shutdown_timeout* 參數。

async asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, ***, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

🄵動一個 Unix socket 伺服器。

與*start_server()* 相似，但會是操作 Unix sockets。

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

另請參🄵*loop.create_unix_server()* 文件。

備🄵

The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's *close()* method.

可用性: Unix.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 與 *start_serving* 參數。*path* 參數現在可以是個*path-like object*。

在 3.10 版的變更: 移除 *loop* 參數。

在 3.11 版的變更: 新增 *ssl_shutdown_timeout* 參數。

在 3.13 版的變更: 新增 *cleanup_socket* 參數。

StreamReader

class asyncio.**StreamReader**

表示一個有提供 API 來從 IO 串流中讀取資料的 reader 物件。作🄵一個*asynchronous iterable*，此物件支援 `async for` 陳述式。

不建議直接實例化 *StreamReader* 物件；使用*open_connection()* 和*start_server()* 會是較好的做法。

feed_eof()

Acknowledge the EOF.

async read(*n=-1*)

從串流中讀取至多 *n* 個位元組的資料。

如果🄵有設定 *n* 或是被設🄵 `-1`，則會持續讀取直到 EOF，然後回傳所有讀取到的*bytes*。讀取到 EOF 且🄵部緩衝區是空的，則回傳一個空的 `bytes` 物件。

如果 *n* 🄵 `0`，則立即回傳一個空的 `bytes` 物件。

If *n* is positive, return at most *n* available `bytes` as soon as at least 1 byte is available in the internal buffer. If EOF is received before any byte is read, return an empty `bytes` object.

async readline()

讀取一行，其中" 行" 指的是以 `\n` 結尾的位元組序列。

如果讀取到 EOF 而🄵有找到 `\n`，該方法會回傳部分的已讀取資料。

如果讀取到 EOF 且🄵部緩衝區是空的，則回傳一個空的 `bytes` 物件。

async readexactly(*n*)

讀取剛好 *n* 個位元組。

如果在讀取完 *n* 個位元組之前讀取到 EOF，則會引發*IncompleteReadError*。使用*IncompleteReadError.partial* 屬性來取得串流結束前已讀取的部分資料。

async readuntil(*separator=b'\n'*)

從串流中持續讀取資料直到出現 *separator*。

成功後，資料和 separator（分隔符號）會從🄵部緩衝區中🄵除（或者🄵是被消費掉 (consumed)）。回傳的資料在末尾會有一個 separator。

如果讀取的資料量超過了設定的串流限制，將會引發*LimitOverrunError* 例外，資料將被留在🄵部緩衝區中，🄵可以再次被讀取。

如果在完整的 separator 被找到之前就讀取到 EOF，則會引發*IncompleteReadError* 例外，且🄵部緩衝區會被重置。*IncompleteReadError.partial* 屬性可能包含一部分的 separator。

The *separator* may also be a tuple of separators. In this case the return value will be the shortest possible that has any separator as the suffix. For the purposes of *LimitOverrunError*, the shortest possible separator is considered to be the one that matched.

在 3.5.2 版被加入.

在 3.13 版的變更: 現在 *separator* 參數可以是一個分隔符號的*tuple*。

at_eof()

如果緩衝區是空的且*feed_eof()* 曾被呼叫則回傳 `True`。

StreamWriter

class asyncio.**StreamWriter**

表示一個有提供 API 來將資料寫入 IO 串流的 writer 物件。

不建議直接實例化 *StreamWriter* 物件；使用*open_connection()* 和*start_server()* 會是較好的做法。

write(*data*)

此方法會嘗試立即將 *data* 寫入到底層的 socket。如果失敗，資料會被放到🄵部寫入緩衝中排隊等待 (queue)，直到它可被發送。

此方法應當與 `drain()` 方法一起使用：

```
stream.write(data)
await stream.drain()
```

> **備🄵**
>
> The *data* buffer should be a C contiguous one-dimensional *bytes-like object*.

writelines(*data*)

此方法會立即嘗試將一個位元組 list（或任何可🄵代物件 (iterable)）寫入到底層的 socket。如果失敗，資料會被放到🄵部寫入緩衝中排隊等待，直到它可被發送。

此方法應當與 `drain()` 方法一起使用：

```
stream.writelines(lines)
await stream.drain()
```

close()

此方法會關閉串流以及底層的 socket。

此方法應與 `wait_closed()` 方法一起使用，但🄵非🄵制：

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

如果底層的傳輸支援*write_eof()* 方法就回傳 `True`，否則回傳 `False`。

write_eof()

在已緩衝的寫入資料被清理 (flush) 後關閉串流的寫入端。

transport

回傳底層的 asyncio 傳輸。

get_extra_info(*name*, *default=None*)

存取可選的傳輸資訊；詳情請見*BaseTransport.get_extra_info()*。

async drain()

等待直到可以繼續寫入到串流。範例：

```
writer.write(data)
await writer.drain()
```

這是一個與底層 IO 寫入緩衝區互動的流程控制方法。當緩衝區大小達到最高標記位 (high watermark) 時，*drain()* 會阻塞直到緩衝區大小🄵少至最低標記位 (low watermark) 以便繼續寫入。當🄵有要等待的資料時，*drain()* 會立即回傳。

async start_tls(*sslcontext*, *, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

將現有的基於串流的連🄵升級到 TLS。

參數：

- *sslcontext*：一個*SSLContext* 的已配置實例。
- *server_hostname*：設定或覆寫將會被目標伺服器憑證比對的主機名稱。
- *ssl_handshake_timeout* is the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

在 3.11 版被加入.

在 3.12 版的變更: 新增 *ssl_shutdown_timeout* 參數。

is_closing()

如果串流已被關閉或正在被關閉則回傳 `True`。

在 3.7 版被加入.

async wait_closed()

等待直到串流被關閉。

應當在*close()* 之後才被呼叫，這會持續等待直到底層的連🄵被關閉，以確保在這之前（例如在程式退出前）所有資料都已經被清空

在 3.7 版被加入.

范例

使用串流的 TCP echo 用🄵端

使用*asyncio.open_connection()* 函式的 TCP echo 用🄵端：

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

也參考

使用低階*loop.create_connection()* 方法的*TCP echo* 用🄵端協定範例。

使用串流的 TCP echo 伺服器

TCP echo 伺服器使用*asyncio.start_server()* 函式：

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

也參考

使用*loop.create_server()* 方法的*TCP echo 伺服器協定* 範例。

取得 HTTP 標頭

查詢自命令列傳入之 URL 所帶有 HTTP 標頭的簡單範例：

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

用法：

```
python example.py http://example.com/path/page.html
```

或使用 HTTPS：

```
python example.py https://example.com/path/page.html
```

🄵🄵一個使用串流來等待資料的開放 socket

等待直到 socket 透過使用*open_connection()* 函式接收到資料的協程：

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
```

(繼續下一頁)

(繼續上一頁)

```
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

也參考

在🄵🄵一個開🄵的 *socket* 以等待有使用協定的資料範例中，有使用了低階協定以及*loop.create_connection()* 方法。

在監視檔案描述器以讀取事件範例中，有使用低階的*loop.add_reader()* 方法來監視檔案描述器。

19.1.4 同步化原始物件 (Synchronization Primitives)

原始碼： Lib/asyncio/locks.py

---

asyncio 的同步化原始物件被設計成和那些*threading* 模組 (module) 中的同名物件相似，但有兩個重要的限制條件：

- asyncio 原始物件🄵不支援執行緒安全 (thread-safe)，因此他們不可被用於 OS 執行緒同步化（請改用*threading*）；
- 這些同步化原始物件的方法 (method) 🄵不接受 *timeout* 引數；要達成有超時 (timeout) 設定的操作請改用*asyncio.wait_for()* 函式。

asyncio 有以下基礎同步化原始物件：

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*
- *Barrier*

Lock

class asyncio.**Lock**

實作了一個給 asyncio 任務 (task) 用的互斥鎖 (mutex lock)。不支援執行緒安全。

一個 asyncio 的鎖可以用來確保一個共享資源的存取權被獨🄴。

使用 Lock 的推薦方式是透過 `async with` 陳述式：

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

這等價於：

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

在 3.10 版的變更: 移除 `loop` 參數。

async acquire()

獲得鎖。

此方法會持續等待直到鎖的狀態成🄵 *unlocked*，🄴將其設置🄵 *locked* 和回傳 `True`。

當多於一個的協程 (coroutine) 在*acquire()* 中等待解鎖而被阻塞，最終只會有其中的一個被處理。

鎖的取得方式是公平的：被處理的協程會是最早開始等待解鎖的那一個。

release()

釋放鎖。

如果鎖的狀態🄵 *locked* 則將其重置🄵 *unlocked* 🄴回傳。

如果鎖的狀態🄵 *unlocked* 則*RuntimeError* 會被引發。

locked()

如果鎖的狀態🄵 *locked* 則回傳 `True`。

Event

class asyncio.**Event**

一個事件 (event) 物件。不支援執行緒安全。

一個 asyncio 事件可以被用於通知多個有發生某些事件於其中的 asyncio 任務。

一個 Event 物件會管理一個🄴部旗標 (flag)，它可以透過*set()* 方法來被設🄵 *true* 🄴透過*clear()* 方法來重置🄵 *false*。*wait()* 方法會被阻塞 (block) 直到該旗標被設🄵 *true*。該旗標初始設置🄵 *false*。

在 3.10 版的變更: 移除 `loop` 參數。 範例：

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')
```

(繼續下一頁)

(繼續上一頁)

```
async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

async wait()

持續等待直到事件被設置。

如果事件有被設置則立刻回傳 `True`。否則持續阻塞直到另一個任務呼叫*set()*。

set()

設置事件。

所有正在等待事件被設置的任務會立即被🄵醒。

clear()

清除（還原）事件。

正透過*wait()* 等待的後續 Tasks 現在會持續阻塞直到*set()* 方法再次被呼叫。

is_set()

如果事件有被設置則回傳 `True`。

Condition

class asyncio.**Condition**(*lock=None*)

一個條件 (codition) 物件。不支援執行緒安全。

一個 asyncio 條件原始物件可以被任務用來等待某事件發生，🄵獲得一個共享資源的獨🄵存取權。

本質上，一個 Condition 物件會結合*Event* 和*Lock* 的功能。多個 Condition 物件共享一個 Lock 是有可能發生的，這能🄵協調關注同一共享資源的不同狀態以取得其獨🄵存取權的多個任務。

可選的 *lock* 引數必須是一個*Lock* 物件或者🄵 `None`。如🄵後者則一個新的 Lock 物件會被自動建立。

在 3.10 版的變更: 移除 `loop` 參數。

使用 Condition 的推薦方式是透過 `async with` 陳述式:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

這等價於:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
```

(繼續下一頁)

(繼續上一頁)

```
    await cond.wait()
finally:
    cond.release()
```

async acquire()

取得底層的鎖。

此方法會持續等待直到底層的鎖🄵 *unlocked*，🄵將其設🄵 *locked* 🄵回傳 `True`。

notify(*n=1*)

🄵醒至多 *n* 個正在等待此條件的任務（預設🄵 1），如果少於 *n* 個任務則全部被🄵醒。

在此方法被呼叫前必須先獲得鎖，🄵在之後立刻將其釋放。如果呼叫於一個 *unlocked* 的鎖則*RuntimeError* 錯誤會被引發。

locked()

如果已取得底層的鎖則回傳 `True`。

notify_all()

🄵醒所有正在等待此條件的任務。

這個方法的行🄵就像*notify()*，但會🄵醒所有正在等待的任務。

在此方法被呼叫前必須先獲得鎖，🄵在之後立刻將其釋放。如果呼叫於一個 *unlocked* 的鎖則*RuntimeError* 錯誤會被引發。

release()

釋放底層的鎖。

當叫用於一個未被解開的鎖之上時，會引發一個*RuntimeError*。

async wait()

持續等待直到被通知 (notify)。

當此方法被呼叫時，如果呼叫它的任務還🄵有取得鎖的話，*RuntimeError* 會被引發。

此方法會釋放底層的鎖，然後持續阻塞直到被*notify()* 或*notify_all()* 的呼叫所🄵醒。一但被🄵醒，Condition 會重新取得該鎖且此方法會回傳 `True`。

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to *wait()* again. For this reason, you may prefer to use *wait_for()* instead.

async wait_for(*predicate*)

持續等待直到謂語 (predicate) 成🄵 *true*。

謂語必須是一個結果可被直譯🄵一個 boolean 值的可呼叫物件 (callable)。此方法會重🄵地*wait()* 直到謂語求值結果🄵 *true*。最終的值即🄵回傳值。

Semaphore

class asyncio.**Semaphore**(*value=1*)

一個旗號 (semaphore) 物件。不支援執行緒安全。

一個旗號物件會管理一個🄵部計數器，會在每次呼叫*acquire()* 時🄵少一、每次呼叫*release()* 時增加一。此計數器永遠不會少於零；當*acquire()* 發現它是零時，它會持續阻塞🄵等待某任務呼叫*release()*。

可選的 *value* 引數給定了🄵部計數器的初始值（預設🄵 `1`）。如給定的值少於 `0` 則*ValueError* 會被引發。

在 3.10 版的變更: 移除 `loop` 參數。

使用 Semaphore 的推薦方式是透過 `async with` 陳述式：

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

這等價於：

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

async acquire()

取得一個旗號。

如果🄵部計數器大於零，將其🄵一🄵立刻回傳 `True`。如果🄵零，則持續等待直到*release()* 被呼叫，🄵回傳 `True`。

locked()

如果旗號無法立即被取得則回傳 `True`。

release()

釋放一個旗號，🄵🄵其🄵部的計數器數值增加一。可以把一個正在等待取得旗號的任務叫醒。

和*BoundedSemaphore* 不同，*Semaphore* 允許 `release()` 的呼叫次數多於 `acquire()`。

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*)

一個有界的旗號物件。不支援執行緒安全。

Bounded Semaphore 是*Semaphore* 的另一版本，如果其🄵部的計數器數值增加至大於初始 *value* 值的話，*ValueError* 會在*release()* 時被引發。

在 3.10 版的變更: 移除 `loop` 參數。

Barrier

class asyncio.**Barrier**(*parties*)

一個屏障 (barrier) 物件。不支援執行緒安全。

A barrier is a simple synchronization primitive that allows to block until *parties* number of tasks are waiting on it. Tasks can wait on the *wait()* method and would be blocked until the specified number of tasks end up waiting on *wait()*. At that point all of the waiting tasks would unblock simultaneously.

`async with` can be used as an alternative to awaiting on *wait()*.

The barrier can be reused any number of times.

範例：

```
async def example_barrier():
   # barrier with 3 parties
   b = asyncio.Barrier(3)

   # create 2 new waiting tasks
   asyncio.create_task(b.wait())
   asyncio.create_task(b.wait())
```

(繼續下一頁)

(繼續上一頁)

```
    await asyncio.sleep(0)
    print(b)

    # The third .wait() call passes the barrier
    await b.wait()
    print(b)
    print("barrier passed")

    await asyncio.sleep(0)
    print(b)

asyncio.run(example_barrier())
```

Result of this example is:

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

在 3.11 版被加入.

async wait()

Pass the barrier. When all the tasks party to the barrier have called this function, they are all unblocked simultaneously.

When a waiting or blocked task in the barrier is cancelled, this task exits the barrier which stays in the same state. If the state of the barrier is "filling", the number of waiting task decreases by 1.

The return value is an integer in the range of 0 to `parties-1`, different for each task. This can be used to select a task to do some special housekeeping, e.g.:

```
...
async with barrier as position:
    if position == 0:
        # Only one task prints this
        print('End of *draining phase*')
```

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a task is waiting. It could raise a *CancelledError* if a task is cancelled.

async reset()

Return the barrier to the default, empty state. Any tasks waiting on it will receive the *BrokenBarrierError* exception.

If a barrier is broken it may be better to just leave it and create a new one.

async abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

The number of tasks required to pass the barrier.

n_waiting

The number of tasks currently waiting in the barrier while filling.

broken

A boolean that is `True` if the barrier is in the broken state.

exception asyncio.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

在 3.9 版的變更: 透過 `await lock` 或 `yield from lock` 和/或 `with` 陳述式 (`with await lock`, `with (yield from lock)`) 來取得鎖的方式已被移除。請改用 `async with lock`。

19.1.5 子行程

原始碼： Lib/asyncio/subprocess.py、Lib/asyncio/base_subprocess.py

---

This section describes high-level async/await asyncio APIs to create and manage subprocesses.

Here's an example of how asyncio can run a shell command and obtain its result:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

會印出:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel. It is indeed trivial to modify the above example to run several commands simultaneously:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

另請參🄵*Examples*。

建立子行程

async asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

建立一個子行程。

The *limit* argument sets the buffer limit for *StreamReader* wrappers for *stdout* and *stderr* (if *subprocess.PIPE* is passed to *stdout* and *stderr* arguments).

回傳一個`Process`實例。

See the documentation of `loop.subprocess_exec()` for other parameters.

在 3.10 版的變更: 移除了 *loop* 參數。

async asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

執行 *cmd* shell 命令。

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `stdout` and `stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

回傳一個`Process`實例。

See the documentation of `loop.subprocess_shell()` for other parameters.

重要

It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special shell characters in strings that are going to be used to construct shell commands.

在 3.10 版的變更: 移除了 *loop* 參數。

備🄵

Subprocesses are available for Windows if a `ProactorEventLoop` is used. See *Subprocess Support on Windows* for details.

也參考

asyncio also has the following *low-level* APIs to work with subprocesses: `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, as well as the *Subprocess Transports* and *Subprocess Protocols*.

常數

asyncio.subprocess.**PIPE**

Can be passed to the *stdin*, *stdout* or *stderr* parameters.

If *PIPE* is passed to *stdin* argument, the `Process.stdin` attribute will point to a `StreamWriter` instance.

If *PIPE* is passed to *stdout* or *stderr* arguments, the `Process.stdout` and `Process.stderr` attributes will point to `StreamReader` instances.

asyncio.subprocess.**STDOUT**

Special value that can be used as the *stderr* argument and indicates that standard error should be redirected into standard output.

asyncio.subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to process creation functions. It indicates that the special file `os.devnull` will be used for the corresponding subprocess stream.

與子行程互動

Both `create_subprocess_exec()` and `create_subprocess_shell()` functions return instances of the *Process* class. *Process* is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

class asyncio.subprocess.**Process**

An object that wraps OS processes created by the `create_subprocess_exec()` and `create_subprocess_shell()` functions.

This class is designed to have a similar API to the `subprocess.Popen` class, but there are some notable differences:

- unlike Popen, Process instances do not have an equivalent to the `poll()` method;
- the `communicate()` and `wait()` methods don't have a *timeout* parameter: use the `wait_for()` function;
- the `Process.wait()` method is asynchronous, whereas `subprocess.Popen.wait()` method is implemented as a blocking busy loop;
- the *universal_newlines* parameter is not supported.

This class is *not thread safe*.

See also the *Subprocess and Threads* section.

async wait()

Wait for the child process to terminate.

Set and return the `returncode` attribute.

> **備🄵**
>
> This method can deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates so much output that it blocks waiting for the OS pipe buffer to accept more data. Use the `communicate()` method when using pipes to avoid this condition.

async communicate(*input=None*)

Interact with process:

1. send data to *stdin* (if *input* is not `None`);
2. 關閉 *stdin*;
3. 從 *stdout* 和 *stderr* 讀取資料，直到 EOF;
4. 等待行程終止。

The optional *input* argument is the data (`bytes` object) that will be sent to the child process.

回傳一個 `(stdout_data, stderr_data)` 元組。

If either `BrokenPipeError` or `ConnectionResetError` exception is raised when writing *input* into *stdin*, the exception is ignored. This condition occurs when the process exits before all data are written into *stdin*.

If it is desired to send data to the process' *stdin*, the process needs to be created with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, the process has to be created with `stdout=PIPE` and/or `stderr=PIPE` arguments.

Note, that the data read is buffered in memory, so do not use this method if the data size is large or unlimited.

在 3.12 版的變更: *stdin* gets closed when `input=None` too.

send_signal(*signal*)

Sends the signal *signal* to the child process.

> **備🄵**
>
> On Windows, *SIGTERM* is an alias for *terminate()*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes CREATE_NEW_PROCESS_GROUP.

terminate()

Stop the child process.

On POSIX systems this method sends *SIGTERM* to the child process.

On Windows the Win32 API function TerminateProcess() is called to stop the child process.

kill()

Kill the child process.

On POSIX systems this method sends *SIGKILL* to the child process.

On Windows this method is an alias for *terminate()*.

stdin

Standard input stream (*StreamWriter*) or None if the process was created with stdin=None.

stdout

Standard output stream (*StreamReader*) or None if the process was created with stdout=None.

stderr

Standard error stream (*StreamReader*) or None if the process was created with stderr=None.

> **警告**
>
> Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read()*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid

Process identification number (PID).

Note that for processes created by the *create_subprocess_shell()* function, this attribute is the PID of the spawned shell.

returncode

Return code of the process when it exits.

A None value indicates that the process has not terminated yet.

A negative value -N indicates that the child was terminated by signal N (POSIX only).

子行程與執行緒

Standard asyncio event loop supports running subprocesses from different threads by default.

On Windows subprocesses are provided by *ProactorEventLoop* only (default), *SelectorEventLoop* has no subprocess support.

Note that alternative event loop implementations might have own limitations; please refer to their documentation.

也參考

The *Concurrency and multithreading in asyncio* section.

范例

An example using the `Process` class to control a subprocess and the `StreamReader` class to read from its standard output.

The subprocess is created by the `create_subprocess_exec()` function:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using low-level APIs.

19.1.6 🄵列 (Queues)

原始碼： Lib/asyncio/queues.py

---

asyncio 🄵列被設計成與`queue` 模組類似。🄵管 asyncio 🄵列不支援執行緒安全 (thread-safe)，但他們是被設計來專用於 async/await 程式。

注意 asyncio 的🄵列🄵有 *timeout* 參數；請使用`asyncio.wait_for()` 函式來🄵🄵列新增具有超時 (timeout) 設定的操作。

另請參🄵下方*Examples*。

Queue

class asyncio.**Queue**(*maxsize=0*)

先進先出 (FIFO) 🄵列。

如果 *maxsize* 小於或等於零，則🄵列大小是無限制的。如果是大於 `0` 的整數，則當🄵列達到 *maxsize* 時，`await put()` 將會阻塞 (block)，直到某個元素被`get()` 取出。

不像標准函式庫中執行緒類型的`queue`，🄵列的大小一直是已知的，可以透過呼叫`qsize()` 方法回傳。

在 3.10 版的變更: 移除 *loop* 參數。

這個類🄵是不支援執行緒安全的。

maxsize

🄵列中可存放的元素數量。

empty()

如果🄵列🄴空則回傳 `True`，否則回傳 `False`。

full()

如果有*maxsize* 個條目在🄵列中，則回傳 `True`。

如果🄵列用 `maxsize=0` （預設）初始化，則*full()* 永遠不會回傳 `True`。

async get()

從🄵列中🄴除🄵回傳一個元素。如果🄵列🄴空，則持續等待直到🄵列中有元素。

Raises *QueueShutDown* if the queue has been shut down and is empty, or if the queue has been shut down immediately.

get_nowait()

如果🄵列🄴有值則立即回傳🄵列中的元素，否則引發*QueueEmpty*。

async join()

持續阻塞直到🄵列中所有的元素都被接收和處理完畢。

當條目新增到🄵列的時候，未完成任務的計數就會增加。每當一個消耗者 (consumer) 協程呼叫*task_done()*，表示這個條目已經被取回且被它包含的所有工作都已完成，未完成任務計數就會🄵少。當未完成計數降到零的時候，*join()* 阻塞會被解除 (unblock)。

async put(*item*)

將一個元素放進🄵列。如果🄵列滿了，在新增元素之前，會持續等待直到有空🄵插槽 (free slot) 能被使用。

如果🄵列已經被關閉，則引發*QueueShutDown*。

put_nowait(*item*)

不阻塞地將一個元素放入🄵列。

如果🄵有立即可用的空🄵插槽，引發*QueueFull*。

qsize()

回傳🄵列中的元素數量。

shutdown(*immediate=False*)

Shut down the queue, making *get()* and *put()* raise *QueueShutDown*.

By default, *get()* on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make *get()* raise immediately instead.

All blocked callers of *put()* and *get()* will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of *join()*.

在 3.13 版被加入.

task_done()

表示前面一個排隊的工作項目已經完成。

由🄵列消耗者使用。對於每個用於取得一個工作項目的*get()*，接續的*task_done()* 呼叫會告訴🄵列這個工作項目的處理已經完成。

如果*join()* 目前正在阻塞，在所有項目都被處理後會解除阻塞（意味著每個以*put()* 放進🄵列的條目都會收到一個*task_done()*）。

`shutdown(immediate=True)` calls *task_done()* for each remaining item in the queue.

如果被呼叫的次數多於放入🄵列中的項目數量，將引發*ValueError*。

Priority Queue（優先🄵列）

class asyncio.**PriorityQueue**

Queue 的變形；按優先順序取出條目 (最小的先取出)。

條目通常是 `(priority_number, data)` 形式的 tuple（元組）。

LIFO Queue

class asyncio.**LifoQueue**

Queue 的變形，先取出最近新增的條目（後進先出）。

例外

exception asyncio.**QueueEmpty**

當🄵列🄵空的時候，呼叫*get_nowait()* 方法會引發這個例外。

exception asyncio.**QueueFull**

當🄵列中條目數量已經達到它的 *maxsize* 時，呼叫*put_nowait()* 方法會引發這個例外。

exception asyncio.**QueueShutDown**

Exception raised when *put()* or *get()* is called on a queue which has been shut down.

在 3.13 版被加入.

范例

🄵列能被用於多個🄵行任務的工作分配：

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
```

(繼續下一頁)

(繼續上一頁)

```
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.7 例外

原始碼： Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**

TimeoutError 的一個已被🄵用的🄵名，當操作已超過規定的截止時間時被引發。

在 3.11 版的變更: 此 class 是*TimeoutError* 的一個🄵名。

exception asyncio.**CancelledError**

該操作已被取消。

當 asyncio Task 被取消時，可以捕獲此例外以執行客🄵化操作。在幾乎所有情🄵下，該例外必須重新被引發。

在 3.8 版的變更: *CancelledError* 現在是*BaseException* 而非*Exception* 的子類🄵。

exception asyncio.**InvalidStateError**

Task 或*Future* 的無效🄵部狀態。

可以在像是🄵已設定結果值的 *Future* 物件設定結果值的情🄵下引發。

exception asyncio.**SendfileNotAvailableError**

”sendfile” 系統呼叫不適用於給定的 socket 或檔案型🄵。

一個*RuntimeError* 的子類🄵。

exception asyncio.**IncompleteReadError**

請求的讀取操作未全部完成。

由*asyncio 串流 APIs* 引發。

此例外是*EOFError* 的子類🄵。

expected

預期的位元組總數 (*int*)。

partial

串流結束之前讀取的*bytes* 字串。

exception asyncio.**LimitOverrunError**

在查詢分隔符號 (separator) 時達到緩衝區 (buffer) 大小限制。

由*asyncio 串流 APIs* 引發。

consumed

要消耗的位元組總數。

19.1.8 Call Graph Introspection

原始碼： Lib/asyncio/graph.py

---

asyncio has powerful runtime call graph introspection utilities to trace the entire call graph of a running *coroutine* or *task*, or a suspended *future*. These utilities and the underlying machinery can be used from within a Python program or by external profilers and debuggers.

在 3.14 版被加入.

asyncio.**print_call_graph**(*future=None*, */*, ***, *file=None*, *depth=1*, *limit=None*)

Print the async call graph for the current task or the provided *Task* or *Future*.

This function prints entries starting from the top frame and going down towards the invocation point.

The function receives an optional *future* argument. If not passed, the current running task will be used.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

If the optional keyword-only *limit* argument is provided, each call stack in the resulting graph is truncated to include at most `abs(limit)` entries. If *limit* is positive, the entries left are the closest to the invocation point. If *limit* is negative, the topmost entries are left. If *limit* is omitted or `None`, all entries are present. If *limit* is `0`, the call stack is not printed at all, only "awaited by" information is printed.

If *file* is omitted or `None`, the function will print to *sys.stdout*.

範例：

以下 Python 程式碼：

```
import asyncio

async def test():
    asyncio.print_call_graph()

async def main():
    async with asyncio.TaskGroup() as g:
        g.create_task(test(), name='test')

asyncio.run(main())
```

會印出：

```
* Task(name='test', id=0x1039f0fe0)
+ Call stack:
|   File 't2.py', line 4, in async test()
+ Awaited by:
   * Task(name='Task-1', id=0x103a5e060)
      + Call stack:
      |   File 'taskgroups.py', line 107, in async TaskGroup.__aexit__()
      |   File 't2.py', line 7, in async main()
```

asyncio.**format_call_graph**(*future=None*, */*, *, *depth=1*, *limit=None*)

Like *print_call_graph()*, but returns a string. If *future* is None and there's no current task, the function returns an empty string.

asyncio.**capture_call_graph**(*future=None*, */*, *, *depth=1*, *limit=None*)

Capture the async call graph for the current task or the provided *Task* or *Future*.

The function receives an optional *future* argument. If not passed, the current running task will be used. If there's no current task, the function returns None.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

會回傳一個 FutureCallGraph 資料類🄴物件：

- FutureCallGraph(future, call_stack, awaited_by)

  Where *future* is a reference to a *Future* or a *Task* (or their subclasses.)

  call_stack 是一個由 FrameCallGraphEntry 物件組成的元組。

  awaited_by 是一個由 FutureCallGraph 物件組成的元組。

- FrameCallGraphEntry(frame)

  Where *frame* is a frame object of a regular Python function in the call stack.

低階工具函式

To introspect an async call graph asyncio requires cooperation from control flow structures, such as *shield()* or *TaskGroup*. Any time an intermediate *Future* object with low-level APIs like *Future.add_done_callback()* is involved, the following two functions should be used to inform asyncio about how exactly such intermediate future objects are connected with the tasks they wrap or control.

asyncio.**future_add_to_awaited_by**(*future*, *waiter*, */*)

Record that *future* is awaited on by *waiter*.

Both *future* and *waiter* must be instances of *Future* or *Task* or their subclasses, otherwise the call would have no effect.

A call to future_add_to_awaited_by() must be followed by an eventual call to the *future_discard_from_awaited_by()* function with the same arguments.

asyncio.**future_discard_from_awaited_by**(*future*, *waiter*, */*)

Record that *future* is no longer awaited on by *waiter*.

Both *future* and *waiter* must be instances of *Future* or *Task* or their subclasses, otherwise the call would have no effect.

19.1.9 事件🄵圈

原始碼： Lib/asyncio/events.py、Lib/asyncio/base_events.py

前言

事件🄵圈是每個 asyncio 應用程式的核心。事件🄵圈執行非同步任務和回呼、執行網路 IO 操作🄵🄵動子行程。

應用程式開發人員通常應使用高階的 asyncio 函式，例如*asyncio.run()*，🄵且很少需要參照事件🄵圈物件或呼叫其方法。本節主要針對那些需要更細粒度控制事件🄵圈行🄵的低階程式碼、函式庫和框架的作者。

取得事件🄵圈

以下的低階函式可用於取得、設置或建立事件🄵圈：

asyncio.**get_running_loop**()

在目前作業系統執行緒中回傳正在運行的事件🄵圈。

如果🄵有運行的事件🄵圈，則引發*RuntimeError*。

此函式只能從協程或回呼函式中呼叫。

在 3.7 版被加入.

asyncio.**get_event_loop**()

取得目前的事件🄵圈。

當從協程或回呼函式呼叫此函式（例如，使用 call_soon 或類似的 API 於排程呼叫），此函式將永遠回傳正在運行的事件🄵圈。

如果🄵有設定正在運行的事件🄵圈，該函式將回傳 `get_event_loop_policy().get_event_loop()` 呼叫的結果。

由於此函式具有相當🄵雜的行🄵（尤其是在使用自訂事件🄵圈策略時），在協程和回呼函式中，建議使用*get_running_loop()* 函式，而不是*get_event_loop()*。

如上所述，可以考慮使用高階的*asyncio.run()* 函式，而不是使用這些較低階的函式手動建立和關閉事件🄵圈。

在 3.14 版的變更: 如果🄵有目前的事件🄵圈，則引發*RuntimeError*。

> **備🄵**
>
> The `asyncio` policy system is deprecated and will be removed in Python 3.16; from there on, this function will return the current running event loop if present else it will return the loop set by *set_event_loop()*.

asyncio.**set_event_loop**(*loop*)

將 *loop* 設置🄵目前 OS 執行緒的目前事件🄵圈。

asyncio.**new_event_loop**()

建立🄵回傳新的事件🄵圈物件。

請注意*get_event_loop()*、*set_event_loop()* 和*new_event_loop()* 函式的行🄵可以透過*設定自訂事件🄵圈策略*進行調整。

目🄵

本頁文件包含以下章節：

- *事件🄵圈方法*章節是事件🄵圈 API 們的參照文件；
- *回呼處理*章節記🄵了從排程方法（如*loop.call_soon()* 和*loop.call_later()*）回傳的*Handle* 和*TimerHandle* 實例；
- *Server 物件*章節記🄵了從事件🄵圈方法（如*loop.create_server()*）回傳的資料型🄵；
- *事件🄵圈實作*章節記🄵了*SelectorEventLoop* 和*ProactorEventLoop* 類🄵；
- *範例*章節展示了如何使用一些事件🄵圈 API。

事件🄵圈方法

事件🄵圈提供以下**低階** API：

- 🄵動和停止🄵圈
- 排程回呼函式
- 排程延遲的回呼函式
- 建立 *Futures* 和 *Tasks*
- 打開網路連🄵
- 建立網路伺服器
- 傳輸檔案
- *TLS* 升級
- 監視檔案描述器
- 直接使用 *socket* 物件
- *DNS*
- 使用管道
- *Unix* 訊號
- 在執行緒池或行程池中執行程式碼
- 錯誤處理 *API*
- 🄵用除錯模式
- 運行子行程

🄵動和停止🄵圈

loop.**run_until_complete**(*future*)

運行直到 *future* （一個*Future* 實例）完成。

如果引數是*協程物件*，則它將被隱式排程🄵*asyncio.Task* 運行。

回傳 Future 的結果或引發其例外。

loop.**run_forever**()

運行事件🄵圈直到*stop()* 被呼叫。

如果在呼叫*run_forever()* 之前呼叫*stop()*，則🄵圈將使用超時🄵零的方式輪詢 I/O 選擇器，運行所有回應 I/O 事件（以及已經排程的事件）的回呼，然後退出。

如果在*run_forever()* 運行時呼叫*stop()*，則🄵圈將運行目前批次的回呼函式，然後退出。請注意，由回呼函式排程的新回呼在此情🄵下不會運行；而是在下次呼叫*run_forever()* 或*run_until_complete()* 時運行。

loop.**stop**()

停止事件🄵圈。

loop.**is_running**()

如果事件🄵圈目前正在運行，則回傳 `True`。

loop.**is_closed**()

如果事件🄵圈已關閉，則回傳 `True`。

loop.**close**()

關閉事件🄵圈。

不得於🄵圈運行中呼叫此函式。將🄵🄵任何待處理的回呼。

此方法清除所有🄵列🄵關閉執行器，但不等待執行器完成。

此方法是🄵等且不可逆的。在事件🄵圈關閉後不應呼叫其他方法。

async loop.**shutdown_asyncgens**()

排程所有目前打開的非同步🄵生器物件使用 aclose() 呼叫來關閉。呼叫此方法後，如果🄵代新的非同步🄵生器，事件🄵圈將發出警告。應該使用此方法可靠地完成所有已排程的非同步🄵生器。

請注意，使用*asyncio.run()* 時不需要呼叫此函式。

範例：

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

在 3.6 版被加入.

async loop.**shutdown_default_executor**(*timeout=None*)

排程預設執行器的關閉，🄵等待它加入*ThreadPoolExecutor* 中的所有執行緒。一旦呼叫了此方法，使用預設執行器與*loop.run_in_executor()* 將引發*RuntimeError*。

timeout 參數指定執行器完成加入所需的時間（以*float* 秒🄵單位）。預設情🄵下🄵 None，不會限制執行器所花費的時間。

如果達到 *timeout*，將發出*RuntimeWarning* 警告，預設執行器將立即終止，不等待其執行緒完成加入。

備🄵

使用*asyncio.run()* 時請勿呼叫此方法，因🄵後者會自動處理預設執行器的關閉。

在 3.9 版被加入.

在 3.12 版的變更: 加入 *timeout* 參數。

排程回呼函式

loop.**call_soon**(*callback*, **args*, *context=None*)

在事件🄵圈的下一次🄵代中排程以 *args* 引數呼叫 *callback* *callback*。

回傳*asyncio.Handle* 的實例，稍後可以用於取消回呼函式。

回呼函式按照其🄵🄵的順序呼叫。每個回呼函式將被呼叫恰好一次。

選用的僅限關鍵字引數 *context* 指定了要給 *callback* 執行的自定義*contextvars.Context*。當未提供 *context* 時，回呼函式使用目前情境。

與*call_soon_threadsafe()* 不同，此方法不是執行緒安全的。

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)

這是*call_soon()* 的執行緒安全變體。當從另一個執行緒排程回呼函式時，必須使用此函式，因🄵*call_soon()* 不是執行緒安全的。

This function is safe to be called from a reentrant context or signal handler, however, it is not safe or fruitful to use the returned handle in such contexts.

如果在已關閉的🄵圈上呼叫，則引發*RuntimeError*。在主應用程式關閉時，這可能發生在次要執行緒上。

請參🄵文件的🄵行和多執行緒部分。

在 3.7 版的變更: 新增了 *context* 僅限關鍵字參數。詳細資訊請參🄵 **PEP 567**。

備🄵

大多數*asyncio*排程函式不允許傳遞關鍵字引數。要傳遞關鍵字引數，請使用*functools.partial()*：

```
# 將會排程 "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

通常使用 partial 物件比使用 lambda 更方便，因🄵 asyncio 可以在除錯和錯誤訊息中更好地呈現 partial 物件。

排程延遲的回呼函式

事件🄵圈提供🄵回呼函式排程在將來某個時間點才呼叫的機制。事件🄵圈使用了單調時鐘來追🄵時間。

`loop.`**`call_later`**(*delay*, *callback*, **args*, *context=None*)

排程 *callback* 在給定的 *delay* 秒數後呼叫（可以是整數或浮點數）。

回傳*asyncio.TimerHandle* 的實例，可用於取消回呼函式。

callback 將只被呼叫恰好一次。如果有兩個回呼函式被排程在完全相同的時間，則其呼叫順序是不定的。

可選的位置引數 *args* 將在呼叫回呼函式時傳遞。如果要使用關鍵字引數呼叫回呼函數，請使用*functools.partial()*。

可選的僅限關鍵字 *context* 引數允許🄵 *callback* 指定自定義的*contextvars.Context* 以提供運行。當未提供 *context* 時，將使用目前情境。

備🄵

For performance, callbacks scheduled with *loop.call_later()* may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

在 3.7 版的變更: 新增了 *context* 僅限關鍵字參數。詳細資訊請參🄵 **PEP 567**。

在 3.8 版的變更: 在 Python 3.7 及更早版本中，使用預設事件🄵圈實作時，*delay* 不能超過一天。這在 Python 3.8 中已經修復。

`loop.`**`call_at`**(*when*, *callback*, **args*, *context=None*)

排程 *callback* 在給定的🄵對時間戳 *when*（整數或浮點數）處呼叫，使用與*loop.time()* 相同的時間參照。

此方法的行🄵與*call_later()* 相同。

回傳*asyncio.TimerHandle* 的實例，可用於取消回呼函式。

備🄵

For performance, callbacks scheduled with *loop.call_at()* may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

在 3.7 版的變更: 新增了 *context* 僅限關鍵字參數。詳細資訊請參🄵 **PEP 567**。

在 3.8 版的變更: 在 Python 3.7 及更早版本中，使用預設事件🄵圈實作時，*when* 和目前時間之間的差值不能超過一天。這在 Python 3.8 中已經修復。

`loop.`**`time`**()

根據事件🄵圈的🄵部單調時鐘，回傳目前時間，以*float* 值表示。

備🄵

在 3.8 版的變更: 在 Python 3.7 及更早版本中，超時（相對 *delay* 或🄵對 *when*）不應超過一天。這在 Python 3.8 中已經修復。

也參考

函式 *asyncio.sleep()*。

建立 Futures 和 Tasks

loop.**create_future**()

建立附加到事件🄵圈的 *asyncio.Future* 物件。

這是在 asyncio 中建立 Futures 的首選方式。這允許第三方事件🄵圈提供 Future 物件的替代實作（具有更好的性能或儀器計測表現）。

在 3.5.2 版被加入.

loop.**create_task**(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

排程執行協程 *coro*。回傳 *Task* 物件。

第三方事件🄵圈可以使用其自己的 *Task* 子類🄵以實現互操作性（interoperability）。在這種情🄵下，結果類型是 *Task* 的子類🄵。

The full function signature is largely the same as that of the *Task* constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

如果提供了 *name* 引數且不🄵 `None`，則將其設置🄵任務的名稱，使用 *Task.set_name()*。

可選的僅限關鍵字 *context* 引數允許🄵 *coro* 指定自定義的 *contextvars.Context* 以提供運行。當未提供 *context* 時，將建立目前情境的副本。

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by *loop.set_task_factory()* will be used.

在 3.8 版的變更: 加入 *name* 參數。

在 3.11 版的變更: 加入 *context* 參數。

在 3.13.3 版的變更: Added `kwargs` which passes on arbitrary extra parameters, including `name` and `context`.

在 3.13.4 版的變更: Rolled back the change that passes on *name* and *context* (if it is None), while still passing on other arbitrary keyword arguments (to avoid breaking backwards compatibility with 3.13.3).

在 3.14 版的變更: All *kwargs* are now passed on. The *eager_start* parameter works with eager task factories.

loop.**set_task_factory**(*factory*)

設置將由 *loop.create_task()* 使用的任務工廠。

如果 *factory* 🄵 `None`，將設置預設的任務工廠。否則，*factory* 必須是一個具有匹配簽名 `(loop, coro, **kwargs)` 的 *callable*，其中 *loop* 是有效事件🄵圈的參照、*coro* 是一個協程物件。該可呼叫物件必須傳遞所有 *kwargs* 🄵回傳一個與 *asyncio.Task* 相容的物件。

在 3.13.3 版的變更: Required that all *kwargs* are passed on to *asyncio.Task*.

在 3.13.4 版的變更: *name* is no longer passed to task factories. *context* is no longer passed to task factories if it is `None`.

在 3.14 版的變更: *name* and *context* are now unconditionally passed on to task factories again.

loop.**get_task_factory**()

回傳任務工廠，如果使用預設任務工廠則回傳 None。

打開網路連🄵

async loop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*, *all_errors=False*)

打開以 *host* 和 *port* 指定之給定地址的串流傳輸連🄵。

根據 *host**（或提供的 **family* 引數）的情🄵，socket 家族可以是*AF_INET* 或*AF_INET6*。

Socket 類型將🄵*SOCK_STREAM*。

protocol_factory 必須是一個回傳*asyncio protocol* 實作的可呼叫函式。

此方法將嘗試在背景建立連🄵。成功時，它將回傳一對 (transport, protocol)。

底層操作的時間軸簡介如下：

1. 建立連🄵🄵🄵其建立傳輸。
2. *protocol_factory* 在無引數的情🄵下被呼叫，🄵且預計回傳一個協定 實例。
3. 通過呼叫其*connection_made()* 方法，將協定實例與傳輸連🄵在一起。
4. 成功時回傳一個 (transport, protocol) 元組。

建立的傳輸是一個依賴實作的雙向串流。

其他引數：

- 若有給定 *ssl* 且非 false，將建立 SSL/TLS 傳輸（預設建立普通 TCP 傳輸）。如果 *ssl* 🄵*ssl.SSLContext* 物件，則使用該情境來建立傳輸；如果 *ssl* 🄵*True*，則使用*ssl.create_default_context()* 回傳的預設情境。

  > **也參考**
  >
  > *SSL/TLS 安全考量*

- *server_hostname* 設置或覆蓋目標伺服器憑證將匹配的主機名稱。僅在 *ssl* 不🄵 None 時傳遞。預設情🄵下，將使用 *host* 引數的值。如果 *host* 🄵空，則🄵有預設值，必須傳遞 *server_hostname* 的值。若 *server_hostname* 🄵空字串，將停用主機名稱匹配（這是一個嚴重的安全風險，可能導致中間人攻擊）。
- *family*、*proto*、*flags* 是可選的位址家族、協定和旗標，用於傳遞至 getaddrinfo() 進行 *host* 解析。若有給定這些應該都是相應*socket* 模組常數的整數。
- 若有給定，*happy_eyeballs_delay* 會🄵用此連🄵的 Happy Eyeballs。它應該是一個浮點數，表示等待連🄵嘗試完成的秒數，然後在🄵行🄵動下一次嘗試。這是 **RFC 8305** 中定義的「連🄵嘗試延遲」。RFC 建議的合理預設值🄵 0.25 秒（250 毫秒）。
- *interleave* 控制主機名稱解析🄵多個 IP 位址時的地址重新排序。若🄵 0 或未指定，將不執行重排序，🄵按*getaddrinfo()* 回傳的順序嘗試位址。如果指定正整數，則按地址家族交錯排列，給定的整數直譯🄵 **RFC 8305** 中定義的「首個地址家族計數」。如果未指定 *happy_eyeballs_delay*，則預設值🄵 0，如果指定則🄵 1。
- 若有給定 *sock* 則其應🄵已存在且已連🄵的*socket.socket* 物件，可供傳輸使用。如果提供了 *sock*，則不應指定 *host*、*port*、*family*、*proto*、*flags*、*happy_eyeballs_delay*、*interleave* 和 *local_addr* 中的任何一項。

> **備🄵**
>
> 引數 *sock* 將 socket 所有權轉移給所建立的傳輸 socket，請呼叫傳輸的`close()` 方法。

- 若有給定 *local_addr* 則其🄵一個 `(local_host, local_port)` 元組，用於在本地綁定 socket。將使用 `getaddrinfo()` 查找 *local_host* 和 *local_port*，方式類似於 *host* 和 *port*。
- *ssl_handshake_timeout*（對於 TLS 連🄵）是等待 TLS 交握的時間，以秒🄵單位，在那之前若未完成則會中斷連🄵。如果🄵 `None`（預設值），則會等待 `60.0` 秒。
- *ssl_shutdown_timeout* 是等待 SSL 關閉完成以前中斷連🄵的時間，以秒🄵單位。如果🄵 `None`（預設值），則會等待 `30.0` 秒。
- *all_errors* 🄵定在無法建立連🄵時會引發哪些例外。預設情🄵下，只會引發單一 `Exception`：如果只有一個例外或所有錯誤訊息相同，則引發第一個例外，否則引發包含所有錯誤訊息的單一 `OSError`。當 `all_errors` 🄵 `True` 時，將引發包含所有例外的 `ExceptionGroup`（即使只有一個例外）。

在 3.5 版的變更: 新增`ProactorEventLoop` 中的 SSL/TLS 支援。

在 3.6 版的變更: 所有 TCP 連🄵都預設有*socket.TCP_NODELAY* socket 選項。

在 3.7 版的變更: 增加 *ssl_handshake_timeout* 參數。

在 3.8 版的變更: 加入 *happy_eyeballs_delay* 和 *interleave* 參數。

Happy Eyeballs 演算法：雙協定堆🄵主機 (Dual-Stack Hosts) 的成功。當伺服器的 IPv4 路徑和協定運作正常，但伺服器的 IPv6 路徑和協定不運作時，雙棧用🄴端應用程式會比僅具 IPv4 的用🄵端體驗到顯著的連🄵延遲。這是不希望的，因🄵這會導致雙棧用🄵端的使用者體驗變差。本文件具體🄵明了🄵少此用🄵可見延遲的演算法要求🄵提供了一種演算法。

更多資訊請見：https://datatracker.ietf.org/doc/html/rfc6555

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

在 3.12 版的變更: 已新增 *all_errors*。

> **也參考**
>
> 函式`open_connection()` 是高階的替代 API。它回傳一對 (`StreamReader`, `StreamWriter`) 可直接在 async/await 程式碼中使用。

async `loop.`**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

建立一個資料報連🄵。

Socket 家族可以是`AF_INET`、`AF_INET6` 或`AF_UNIX`，視乎 *host*（或提供的 *family* 引數）而定。

Socket 類型將🄵`SOCK_DGRAM`。

protocol_factory 必須是可呼叫的函式，回傳*protocol* 實作。

成功時回傳 `(transport, protocol)` 元組。

其他引數：

- *local_addr*，如果提供，是一個 `(local_host, local_port)` 元組，用於在本地綁定 socket。*local_host* 和 *local_port* 使用`getaddrinfo()` 來查找。

> **備🄵**

On Windows, when using the proactor event loop with `local_addr=None`, an *OSError* with `errno.WSAEINVAL` will be raised when running it.

- *remote_addr*，如果提供，是一個 `(remote_host, remote_port)` 元組，用於將 socket 連🄵到遠端位址。*remote_host* 和 *remote_port* 使用*getaddrinfo()* 來查找。
- *family*、*proto* 和 *flags* 是用於傳遞給*getaddrinfo()* 以解析 *host* 的可選地址家族、協定和旗標。如果提供，這些應該都是來自相應的*socket* 模組常數的整數。
- *reuse_port* 告訴核心允許將此端點綁定到與其他現有端點相同的埠，只要它們在建立時都設定了此旗標。此選項不受 Windows 和某些 Unix 系統支援。如果未定義 `SO_REUSEPORT` 常數，則不支援此功能。
- *allow_broadcast* 告訴核心允許此端點向廣播位址發送訊息。
- *sock* 可以選擇性地指定，以使用預先存在且已連🄵的*socket.socket* 物件供傳輸使用。如果指定，*local_addr* 和 *remote_addr* 應省略（必須是*None*）。

備🄵

引數 *sock* 將 socket 所有權轉移給所建立的傳輸 socket，請呼叫傳輸的*close()* 方法。

請參🄵*UDP 回應用🄵端協定* 和*UDP 回應伺服器協定* 範例。

在 3.4.4 版的變更: 新增 *family*、*proto*、*flags*、*reuse_address*、*reuse_port*、*allow_broadcast* 和 *sock* 參數。

在 3.8 版的變更: 新增對於 Windows 的支援。

在 3.8.1 版的變更: 不再支援 *reuse_address* 參數，因🄵使用 `SO_REUSEADDR` 對於 UDP 存有重大的安全疑慮。明確傳遞 `reuse_address=True` 將引發例外。

當具有不同 UID 的多個行程使用 `SO_REUSEADDR` 將 socket 分配給相同的 UDP socket 地址時，傳入的封包可能會在 socket 之間隨機分🄵。

對於有支援的平台，*reuse_port* 可以用作類似功能的替代方案。使用 *reuse_port*，將改🄵使用 `SO_REUSEPORT`，該選項明確禁止具有不同 UID 的行程將 socket 分配給相同的 socket 地址。

在 3.11 版的變更: 自 Python 3.9.0、3.8.1、3.7.6 和 3.6.10 起，已完全移除 *reuse_address* 參數。

async loop.**create_unix_connection**(*protocol_factory*, *path=None*, ***, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

建立一個 Unix 連🄵。

Socket 家族將🄵*AF_UNIX*；socket 類型將🄵*SOCK_STREAM*。

成功時回傳 `(transport, protocol)` 元組。

path 是 Unix 域 socket 的名稱，除非指定 *sock* 參數，否則🄵必填。支援抽象 Unix sockets、*str*、*bytes* 和*Path* 路徑。

有關此方法的引數資訊，請參🄵*loop.create_connection()* 方法的文件。

可用性: Unix.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 參數。*path* 參數現在可以是*path-like object*。

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

建立網路伺服器

async loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, ***, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

建立一個 TCP 伺服器（socket 類型*SOCK_STREAM*），監聽 *host* 位址的 *port*。

回傳一個*Server* 物件。

引數：

- *protocol_factory* 必須是可呼叫的函式，回傳*protocol* 實作。
- 可以將 *host* 參數設🄵幾種類型，以確定伺服器將監聽的位置：
  - 如果 *host* 是字串，則 TCP 伺服器綁定到由 *host* 指定的單個網路介面。
  - 如果 *host* 是字串序列，則 TCP 伺服器綁定到序列指定的所有網路介面。
  - 若 *host* 是空字串或 `None`，則所有介面都被假定🄵回傳多個 socket 的清單（可能一個用於 IPv4，另一個用於 IPv6）。
- 可以設定 *port* 參數以指定伺服器應該監聽的埠。如果是 `0` 或 `None`（預設值），將隨機選擇一個未使用的埠（請注意，如果 *host* 解析🄵多個網路介面，將🄵每個介面隨機選擇不同的隨機埠）。
- *family* 可以設定🄵*socket.AF_INET* 或*AF_INET6* 以🄵制使用 IPv4 或 IPv6。如果未設定，*family* 將從主機名稱🄵定（預設🄵*AF_UNSPEC*）。
- *flags* 是*getaddrinfo()* 的位元遮罩。
- 可以可選地指定 *sock* 以使用現有的 socket 物件。如果指定了，*host* 和 *port* 不能指定。

> **備🄵**
>
> *sock* 引數將 socket 的所有權轉移給建立的伺服器。要關閉 socket，請呼叫伺服器的*close()* 方法。

- *backlog* 是傳遞給*listen()* 的最大🄵列連🄵數（預設🄵 100）。
- *ssl* 可以設定🄵*SSLContext* 實例以在接受的連🄵上🄵用 TLS。
- *reuse_address* 告訴🄵核重用 `TIME_WAIT` 狀態下的本地 socket，而不等待其自然超時過期。如果未指定，在 Unix 上將自動設置🄵 `True`。
- *reuse_port* 告訴🄵核允許此端點🄵結到與其他現有端點🄵結的相同埠，只要它們在建立時都設置了此旗標。此選項在旗標 Windows 上不受支援。
- 將 *keep_alive* 設🄵 `True` 透過🄵用定期的訊息傳輸來保持連🄵活躍。

在 3.13 版的變更: 加入 *keep_alive* 參數。

- （對於 TLS 伺服器）*ssl_handshake_timeout* 是在中斷連🄵之前等待 TLS 握手完成的時間（以秒🄵單位）。如果🄵 `None`（預設），則🄵 `60.0` 秒。
- *ssl_shutdown_timeout* 是等待 SSL 關閉完成以前中斷連🄵的時間，以秒🄵單位。如果🄵 `None`（預設值），則會等待 `30.0` 秒。
- 將 *start_serving* 設置🄵 `True`（預設）將使建立的伺服器立即開始接受連接。當設置🄵 `False` 時，用🄵應該等待*Server.start_serving()* 或*Server.serve_forever()* 來使伺服器開始接受連🄵。

在 3.5 版的變更: 新增*ProactorEventLoop* 中的 SSL/TLS 支援。

在 3.5.1 版的變更: *host* 參數可以是字串序列。

在 3.6 版的變更: 新增 *ssl_handshake_timeout* 與 *start_serving* 參數。所有 TCP 連🄵都預設有*socket.TCP_NODELAY* socket 選項。

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

> **也參考**
>
> *start_server()* 函式是一個更高階的替代 API，它回傳一對*StreamReader* 和*StreamWriter*，可以在 async/await 程式碼中使用。

async loop.**create_unix_server**(*protocol_factory*, *path=None*, *, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

類似*loop.create_server()*，但適用於*AF_UNIX* socket 家族。

path 是 Unix 域 socket 的名稱，除非提供了 *sock* 引數，否則必須給定。支援抽象 Unix sockets、*str*、*bytes* 和*Path* 路徑。

如果 *cleanup_socket* 🄵真，則 Unix socket 將在伺服器關閉時自動從檔案系統中🄵除，除非在建立伺服器後替🄵了 socket。

有關此方法的引數資訊，請參🄵*loop.create_server()* 方法的文件。

可用性: Unix.

在 3.7 版的變更: 新增 *ssl_handshake_timeout* 與 *start_serving* 參數。*path* 參數現在可🄵一個*Path* 物件。

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

在 3.13 版的變更: 加入 *cleanup_socket* 參數。

async loop.**connect_accepted_socket**(*protocol_factory*, *sock*, *, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

將已接受的連🄵包裝成傳輸層/協定對。

此方法可以由在 asyncio 外接受連🄵但使用 asyncio 處理連🄵的伺服器使用。

參數:

- *protocol_factory* 必須是可呼叫的函式，回傳*protocol* 實作。
- *sock* 是從*socket.accept* 回傳的預先存在的 socket 物件。

  > **備🄵**
  >
  > 引數 *sock* 將 socket 所有權轉移給所建立的傳輸 socket，請呼叫傳輸的*close()* 方法。

- *ssl* 可以設置🄵*SSLContext* 以在已接受的連🄵上🄵用 SSL。
- （對於 SSL 連🄵）*ssl_handshake_timeout* 是在中斷連🄵之前等待 SSL 握手完成的時間（以秒🄵單位）。如果🄵 `None`（預設），則🄵 `60.0` 秒。
- *ssl_shutdown_timeout* 是等待 SSL 關閉完成以前中斷連🄵的時間，以秒🄵單位。如果🄵 `None`（預設值），則會等待 `30.0` 秒。

回傳 `(transport, protocol)` 對。

在 3.5.3 版被加入.

在 3.7 版的變更: 增加 *ssl_handshake_timeout* 參數。

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

傳輸檔案

async loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

通過 *transport* 發送 *file*。回傳發送的總位元組數。

如果可用，該方法使用高性能*os.sendfile()*。

file 必須是以二進位模式打開的常規檔案物件。

offset 告訴從哪🄵開始讀取檔案。如果指定了，*count* 是要傳輸的總位元組數，而不是發送檔案直到達到 EOF。即使此方法引發錯誤時，檔案位置也始終更新，可以使用*file.tell()* 取得實際發送的位元組數。

將 *fallback* 設置🄵 `True` 會使 asyncio 在平台不支援 sendfile 系統呼叫時（例如 Windows 或 Unix 上的 SSL socket）手動讀取和發送檔案。

如果系統不支援 *sendfile* 系統呼叫且 *fallback* 🄵 `False`，則引發*SendfileNotAvailableError*。

在 3.7 版被加入.

TLS 升級

async loop.**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

將基於傳輸的現有連🄵升級到 TLS。

建立 TLS 編解碼器實例🄵在 *transport* 和 *protocol* 之間插入它。編解碼器既實作了對於 *transport* 的協定，也實作了對於 *protocol* 的傳輸。

回傳建立的雙介面實例。在 *await* 後，*protocol* 必須停止使用原始的 *transport*，僅與回傳的物件通信，因🄵編碼器快取了 *protocol* 端的資料，🄵且與 *transport* 間歇性地交🄵額外的 TLS session 封包。

在某些情🄵下（例如傳入的傳輸已經關閉），此函式可能回傳 `None`。

參數:

- *transport* 和 *protocol* 實例，由像*create_server()* 和*create_connection()* 等方法回傳。
- *sslcontext*：配置好的*SSLContext* 實例。
- 當升級伺服器端連🄵時（像由*create_server()* 建立的那樣）傳遞 `True`。
- *server_hostname*：設置或覆蓋將用於匹配目標伺服器憑證的主機名。
- *ssl_handshake_timeout*（對於 TLS 連🄵）是等待 TLS 交握的時間，以秒🄵單位，在那之前若未完成則會中斷連🄵。如果🄵 `None`（預設值），則會等待 `60.0` 秒。
- *ssl_shutdown_timeout* 是等待 SSL 關閉完成以前中斷連🄵的時間，以秒🄵單位。如果🄵 `None`（預設值），則會等待 `30.0` 秒。

在 3.7 版被加入.

在 3.11 版的變更: 增加 *ssl_shutdown_timeout* 參數。

監視檔案描述器

loop.**add_reader**(*fd*, *callback*, **args*)

開始監視 *fd* 檔案描述器的讀取可用性，一但 *fd* 可讀取，使用指定引數叫用 *callback*。

任何預先存在、🄵 *fd* 🄵🄵的回呼函式將被取消🄵替🄵🄵 *callback*。

loop.**remove_reader**(*fd*)

停止監視 *fd* 檔案描述器的讀取可用性。如果 *fd* 之前正在監視讀取，則回傳 `True`。

loop.**add_writer**(*fd*, *callback*, **args*)

開始監視 *fd* 檔案描述器的寫入可用性，一旦 *fd* 可寫入，使用指定引數叫用 *callback*。

任何預先存在、🄵 *fd* 🄵🄵的回呼函式將被取消🄵替🄵🄵 *callback*。

使用*functools.partial()* 向 *callback* 傳送關鍵字引數。

loop.**remove_writer**(*fd*)

停止監視 *fd* 檔案描述器的寫入可用性。如果 *fd* 之前正在監視寫入，則回傳 `True`。

另請參🄵平台支援部分以了解這些方法的一些限制。

直接使用 socket 物件

一般情🄵下，使用基於傳輸的 API（如*loop.create_connection()* 和*loop.create_server()*）的協定實作比直接使用 socket 的實作更快。然而在某些情🄵下性能不是關鍵，直接使用*socket* 物件更方便。

async loop.**sock_recv**(*sock*, *nbytes*)

從 *sock* 接收最多 *nbytes*。*socket.recv()* 的非同步版本。

將接收到的資料作🄵 bytes 物件回傳。

sock 必須是非阻塞 socket。

在 3.7 版的變更: 🄵管此方法一直記🄵🄵協程方法，但 Python 3.7 之前的版本回傳*Future*。自 Python 3.7 起，這是 `async def` 方法。

async loop.**sock_recv_into**(*sock*, *buf*)

從 *sock* 接收資料到 *buf* 緩衝區。仿照阻塞*socket.recv_into()* 方法。

回傳寫入緩衝區位元組的數目。

sock 必須是非阻塞 socket。

在 3.7 版被加入.

async loop.**sock_recvfrom**(*sock*, *bufsize*)

從 *sock* 接收最多 *bufsize* 大小的資料單元。*socket.recvfrom()* 的非同步版本。

回傳一個元組 (received data, remote address)。

sock 必須是非阻塞 socket。

在 3.11 版被加入.

async loop.**sock_recvfrom_into**(*sock*, *buf*, *nbytes=0*)

從 *sock* 接收最多 *nbytes* 大小的資料單元到 *buf*。*socket.recvfrom_into()* 的非同步版本。

回傳一個元組 (number of bytes received, remote address)。

sock 必須是非阻塞 socket。

在 3.11 版被加入.

async loop.**sock_sendall**(*sock*, *data*)

將 *data* 發送到 *sock* socket。*socket.sendall()* 的非同步版本。

此方法將繼續發送到 socket，直到 *data* 中的所有資料都已發送或發生錯誤。成功時回傳 `None`。錯誤時引發例外。此外，🄵有辦法確定接收端成功處理了多少資料（如果有的話）。

sock 必須是非阻塞 socket。

在 3.7 版的變更: 🄵管該方法一直被記🄵🄵協程方法，但在 Python 3.7 之前它回傳*Future*。從 Python 3.7 開始，這是一個 `async def` 方法。

async loop.**sock_sendto**(*sock*, *data*, *address*)

從 *sock* 向 *address* 發送一個資料單元。*socket.sendto()* 的非同步版本。

回傳發送的位元組數。

sock 必須是非阻塞 socket。

在 3.11 版被加入.

async loop.**sock_connect**(*sock*, *address*)

將 *sock* 連🄵到位於 *address* 的遠端 socket。

socket.connect() 的非同步版本。

sock 必須是非阻塞 socket。

在 3.5.2 版的變更: 不再需要解析 `address`。`sock_connect` 將嘗試透過呼叫*socket.inet_pton()* 檢查 *address* 是否已解析。如果🄵有，將使用*loop.getaddrinfo()* 解析 *address*。

> **也參考**
>
> *loop.create_connection()* 和*asyncio.open_connection()*。

async loop.**sock_accept**(*sock*)

接受一個連🄵。模擬阻塞的*socket.accept()* 方法。

Socket 必須🄵結到一個地址🄵偵聽連🄵。回傳值是一個 (conn, address) 對，其中 *conn* 是一個 新 socket 物件，可在連🄵上發送和接收資料，*address* 是連接另一端對應的 socket 地址。

sock 必須是非阻塞 socket。

在 3.7 版的變更: 🄵管該方法一直被記🄵🄵協程方法，但在 Python 3.7 之前它回傳*Future*。從 Python 3.7 開始，這是一個 async def 方法。

> **也參考**
>
> *loop.create_server()* 和*start_server()*。

async loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

如果可行，使用高性能*os.sendfile* 發送檔案。回傳發送的總位元組數。

socket.sendfile() 的非同步版本。

sock 必須是非阻塞的*socket.SOCK_STREAM socket*。

file 必須是以二進位模式打開的常規檔案物件。

offset 告訴從哪🄵開始讀取檔案。如果指定了，*count* 是要傳輸的總位元組數，而不是發送檔案直到達到 EOF。即使此方法引發錯誤時，檔案位置也始終更新，可以使用*file.tell()* 取得實際發送的位元組數。

當設置🄵 True 時，*fallback* 使 asyncio 在平台不支援 sendfile 系統呼叫時（例如 Windows 或 Unix 上的 SSL socket）手動讀取和發送檔案。

如果系統不支援 *sendfile* 系統呼叫且 *fallback* 🄵 False，引發*SendfileNotAvailableError*。

sock 必須是非阻塞 socket。

在 3.7 版被加入.

DNS

async loop.**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)

socket.getaddrinfo() 的非同步版本。

async loop.**getnameinfo**(*sockaddr*, *flags=0*)

socket.getnameinfo() 的非同步版本。

> **備🄵**
>
> Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop's default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

在 3.7 版的變更: *getaddrinfo* 和 *getnameinfo* 方法一直被記🄵🄵回傳協程，但在 Python 3.7 之前它們實際上回傳*asyncio.Future* 物件。從 Python 3.7 開始，兩個方法都是協程。

使用管道

async loop.**connect_read_pipe**(*protocol_factory*, *pipe*)

在事件🄴圈中🄵🄴 *pipe* 的讀取端。

protocol_factory 必須是一個回傳*asyncio protocol* 實作的可呼叫函式。

pipe 是*類檔案物件*。

回傳 `(transport, protocol)` 對，其中 *transport* 支援*ReadTransport* 介面，*protocol* 是由 *protocol_factory* 實例化的物件。

使用*SelectorEventLoop* 事件🄴圈時，*pipe* 設置🄵非阻塞模式。

async loop.**connect_write_pipe**(*protocol_factory*, *pipe*)

在事件🄴圈中🄵🄴 *pipe* 的寫入端。

protocol_factory 必須是一個回傳*asyncio protocol* 實作的可呼叫函式。

pipe 是*file-like object*。

回傳 `(transport, protocol)` 對，其中 *transport* 支援*WriteTransport* 介面，*protocol* 是由 *protocol_factory* 實例化的物件。

使用*SelectorEventLoop* 事件🄴圈時，*pipe* 設置🄵非阻塞模式。

備🄵

SelectorEventLoop 在 Windows 上不支援上述方法。對於 Windows 請使用*ProactorEventLoop*。

也參考

loop.subprocess_exec() 和*loop.subprocess_shell()* 方法。

Unix 訊號

loop.**add_signal_handler**(*signum*, *callback*, **args*)

將 *callback* 設置🄵 *signum* 訊號的處理程式。

該回呼將由 *loop* 叫用，與該事件🄴圈的其他排隊回呼和可運行的協程一起。與使用*signal.signal()* 🄵🄵的訊號處理程式不同，使用此函式🄵🄵的回呼允許與事件🄴圈進行互動。

如果訊號號無效或不可捕獲，引發*ValueError*。如果設定處理程序有問題，🄵出*RuntimeError*。

使用*functools.partial()* 向 *callback* *傳送關鍵字引數*。

像*signal.signal()* 一樣，此函式必須在主執行緒中叫用。

loop.**remove_signal_handler**(*sig*)

移除 *sig* 訊號的處理程式。

如果訊號處理程式被移除，回傳 `True`；如果給定訊號🄵有設置處理程式，回傳 `False`。

可用性: Unix.

也參考

signal 模組。

在執行緒池或行程池中執行程式碼

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)

安排在指定的執行器中呼叫 *func*。

The *executor* argument should be an *concurrent.futures.Executor* instance. The default executor is used if *executor* is None. The default executor can be set by *loop.set_default_executor()*, otherwise, a *concurrent.futures.ThreadPoolExecutor* will be lazy-initialized and used by *run_in_executor()* if needed.

範例：

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

    # 4. Run in a custom interpreter pool:
    with concurrent.futures.InterpreterPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom interpreter pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

請注意，由於*multiprocessing*（由*ProcessPoolExecutor* 使用）的特殊性，選項 3 需要進入點保護（if __name__ == '__main__'）。請參🄵主模組的安全引入。

此方法回傳*asyncio.Future* 物件。

使用*functools.partial()* 將來關鍵字引數傳遞給 *func*。

在 3.5.3 版的變更: *loop.run_in_executor()* 不再配置它建立的執行緒池執行器的 `max_workers`，而是讓執行緒池執行器（*ThreadPoolExecutor*）設定預設值。

`loop.`**`set_default_executor`**(*executor*)

將 *executor* 設置🄵*run_in_executor()* 使用的預設執行器。*executor* 必須是*ThreadPoolExecutor*（也包含*InterpreterPoolExecutor*）的實例。

在 3.11 版的變更: *executor* 必須是*ThreadPoolExecutor* 的實例。

錯誤處理 API

允許自定義事件🄵圈中的例外處理方式。

`loop.`**`set_exception_handler`**(*handler*)

將 *handler* 設定🄵新的事件🄵圈例外處理程式。

如果 *handler* 是 `None`，則將設置預設例外處理程式。否則，*handler* 必須是一個可呼叫物件，簽名匹配 `(loop, context)`，其中 `loop` 是參照活躍事件🄵圈的，`context` 是包含例外詳細資訊的 `dict` 物件（有關情境的詳細資訊，請參🄵*call_exception_handler()* 文件）。

如果代表*Task* 或*Handle* 呼叫處理程式，它將在該任務或回呼處理程式的*contextvars.Context* 中運行。

在 3.12 版的變更: 處理程式可能在引發例外的任務或處理程式的*Context* 中被呼叫。

`loop.`**`get_exception_handler`**()

回傳目前的例外處理程式，如果未設置自定義例外處理程式，則回傳 `None`。

在 3.5.2 版被加入.

`loop.`**`default_exception_handler`**(*context*)

預設例外處理程式。

當發生例外且未設置例外處理程式時呼叫此函式。自定義例外處理程式可以呼叫此函式以轉由預設處理程式處理。

context 參數與*call_exception_handler()* 中的意思相同。

`loop.`**`call_exception_handler`**(*context*)

呼叫目前事件🄵圈例外處理程式。

context 是一個包含以下鍵的 `dict` 物件（未來的 Python 版本中可能會引入新的鍵）：

- 'message'：錯誤訊息；
- 'exception'（可選）：例外物件；
- 'future'（可選）：*asyncio.Future* 實例；
- 'task'（可選）：*asyncio.Task* 實例；
- 'handle'（可選）：*asyncio.Handle* 實例；
- 'protocol'（可選）：*Protocol* 實例；
- 'transport'（可選）：*Transport* 實例；
- 'socket'（可選）：*socket.socket* 實例；
- 'source_traceback' (optional): Traceback of the source;
- 'handle_traceback' (optional): Traceback of the handle;
- **'asyncgen'(可選)：非同步🄵生器引發**
  例外。

> **備🄵**
>
> 此方法不應在子類🄵事件🄵圈中被覆寫。🄵了自定義例外處理，請使用*set_exception_handler()* 方法。

🄵用除錯模式

`loop.get_debug`()

取得事件🄵圈的除錯模式（*bool*）。

如果環境變數 `PYTHONASYNCIODEBUG` 被設定🄵非空字串，則預設值🄵 `True`，否則🄵 `False`。

`loop.set_debug`(*enabled:* bool)

設定事件🄵圈的除錯模式。

在 3.7 版的變更: 現在也可以使用新的*Python 開發模式* 🄵用除錯模式。

`loop.slow_callback_duration`

此屬性可用於設定被視🄵" 慢" 的最短執行時間（以秒🄵單位）。🄵用偵錯模式後，" 慢" 回呼將被記🄵。

預設值🄵 100 毫秒

> **也參考**
>
> *asyncio 的除錯模式*。

運行子行程

本小節描述的方法是低階的。在常規的 async/await 程式碼中，請考慮使用高階*asyncio.create_subprocess_shell()* 和*asyncio.create_subprocess_exec()* 輔助功能而不是。

> **備🄵**
>
> 在 Windows 上，預設事件🄵圈*ProactorEventLoop* 支援子行程，而*SelectorEventLoop* 不支援。詳細資訊請參見*Windows 上對於子行程的支援*。

async `loop.subprocess_exec`(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

從 *args* 指定的一個或多個字串引數建立子行程。

args 必須是由以下項表示的字串串列:

- *str*;
- 或*bytes*，編碼🄵*檔案系統編碼*。

第一個字串指定程序可執行檔案，其余字串指定引數。字串引數一起組成程序的 `argv`。

這與標准函式庫*subprocess.Popen* 類似，使用 `shell=False` 呼叫🄵將字串串列作🄵第一個引數傳遞；然而，*Popen* 接受單個字串串列引數，*subprocess_exec* 接受多個字串引數。

protocol_factory 必須是回傳*asyncio.SubprocessProtocol* 子類🄵的可呼叫物件。

其他參數:

- *stdin* 可以是以下任意一個:
  - 類檔案物件
  - 現有的檔案描述器（正整數），例如用*os.pipe()* 建立的

  - *subprocess.PIPE* 常數（預設），它將建立一個新的管道🄵連🄵，
  - 值 None 將使子行程從此行程繼承檔案描述器
  - *subprocess.DEVNULL* 常數，表示將使用特殊的*os.devnull* 檔案
- *stdout* 可以是以下任意一個：
  - 類檔案物件
  - *subprocess.PIPE* 常數（預設），它將建立一個新的管道🄵連🄵，
  - 值 None 將使子行程從此行程繼承檔案描述器
  - *subprocess.DEVNULL* 常數，表示將使用特殊的*os.devnull* 檔案
- *stderr* 可以是以下任意一個：
  - 類檔案物件
  - *subprocess.PIPE* 常數（預設），它將建立一個新的管道🄵連🄵，
  - 值 None 將使子行程從此行程繼承檔案描述器
  - *subprocess.DEVNULL* 常數，表示將使用特殊的*os.devnull* 檔案
  - *subprocess.STDOUT* 常數，它將標準錯誤串流連🄵到行程的標準輸出串流
- 所有其他關鍵字引數都會傳遞給*subprocess.Popen* 而不進行直譯，但 *bufsize*、*universal_newlines*、*shell*、*text*、*encoding* 和 *errors* 除外，這些不應該指定。

  asyncio 子行程 API 不支援將串流解碼🄵文本。可以使用*bytes.decode()* 將從串流回傳的位元組轉🄵🄵文本。

如果傳遞給 *stdin*、*stdout* 或 *stderr* 的類檔案物件表示管道，則該管道的另一端應該使用*connect_write_pipe()* 或*connect_read_pipe()* 🄵🄵到事件🄵圈中。

有關其他引數的文件，請參🄵*subprocess.Popen* 類🄵的建構函式。

回傳 (transport, protocol) 對，其中 *transport* 符合*asyncio.SubprocessTransport* 基底類🄵，*protocol* 是由 *protocol_factory* 實例化的物件。

async loop.**subprocess_shell**(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

使用平台的"shell" 語法從 *cmd* 建立子行程，*cmd* 可以是*str* 或編碼🄵*檔案系統編碼* 的*bytes* 字串。

這類似於標準函式庫中的*subprocess.Popen* 類🄵，使用 shell=True 呼叫。

protocol_factory 必須是回傳*SubprocessProtocol* 子類🄵的可呼叫物件。

有關其余引數的更多詳細資訊，請參🄵*subprocess_exec()*。

回傳一對 (transport, protocol)，其中 *transport* 符合*SubprocessTransport* 基底類🄵，而 *protocol* 是由 *protocol_factory* 實例化的物件。

備🄵

由應用程式負責確保適當引用所有空白和特殊字元，以避免 shell 注入風險。可以使用*shlex.quote()* 函式來正確跳🄵用於構建 shell 命令的字串中的空白和特殊字元。

回呼處理

class asyncio.**Handle**

由*loop.call_soon()* 和*loop.call_soon_threadsafe()* 回傳的回呼包裝器。

get_context()

回傳與處理相關聯的*contextvars.Context* 物件。

在 3.12 版被加入.

cancel()

取消回呼。如果回呼已被取消或執行，此方法將不起作用。

cancelled()

如果回呼已被取消，回傳 `True`。

在 3.7 版被加入.

class asyncio.**TimerHandle**

由*loop.call_later()* 和*loop.call_at()* 回傳的回呼包裝器。

這個類🄵是*Handle* 的子類🄵。

when()

回傳預定的回呼時間，以*float* 秒🄵單位。

時間是一個🄵對的時間戳，使用與*loop.time()* 相同的時間參照。

在 3.7 版被加入.

Server 物件

Server 物件是由*loop.create_server()*、*loop.create_unix_server()*、*start_server()* 和*start_unix_server()* 函式所建立。

請勿直接實例化*Server* 類🄵。

class asyncio.**Server**

Server 物件是非同步情境管理器。當在 `async with` 陳述中使用時，可以保證在完成 `async with` 陳述時，Server 物件將會關閉🄵停止接受新的連🄵：

```
srv = await loop.create_server(...)

async with srv:
    # 一些程式碼

# 此時 srv 已關閉，不再接受新的連🄵。
```

在 3.7 版的變更: 自 Python 3.7 起，Server 物件是非同步情境管理器。

在 3.11 版的變更: 此類🄵在 Python 3.9.11、3.10.3 和 3.11 中以 `asyncio.Server` 的形式被公開。

close()

停止服務：關閉監聽的 sockets 🄵將*sockets* 屬性設🄵 `None`。

代表現有傳入用🄵端連🄵的 sockets 仍然保持開🄵。

伺服器以非同步方式關閉；使用 `wait_close()` 協程等待伺服器關閉（不再有活躍連🄵）。

close_clients()

關閉所有現有的傳入用🄵端連🄵。

在所有關聯的傳輸上呼叫*close()*。

close() should be called before *close_clients()* when closing the server to avoid races with new clients connecting.

在 3.13 版被加入.

abort_clients()

立即關閉所有現有的傳入用🄵端連🄵，而不等待待定操作完成。

在所有關聯的傳輸上呼叫*close()*。

close() should be called before *abort_clients()* when closing the server to avoid races with new clients connecting.

在 3.13 版被加入.

get_loop()

回傳與伺服器物件關聯的事件🄵圈。

在 3.7 版被加入.

async start_serving()

開始接受連🄵。

此方法是冪等的，因此可以在伺服器已經運行時呼叫。

start_serving 僅限關鍵字參數只能在*loop.create_server()* 和*asyncio.start_server()* 中使用，允許建立一個最初不接受連🄵的 Server 物件。在這種情🄵下，可以使用 `Server.start_serving()` 或*Server.serve_forever()* 來使 Server 開始接受連🄵。

在 3.7 版被加入.

async serve_forever()

開始接受連🄵，直到協程被取消。取消 `serve_forever` 任務會導致伺服器關閉。

如果伺服器已經接受連🄵，則可以呼叫此方法。每個 *Server* 物件只能存在一個 `serve_forever` 任務。

範例：

```
async def client_connected(reader, writer):
    # 透過讀取器/寫入器串流
    # 與用🄴端溝通。例如：
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

在 3.7 版被加入.

is_serving()

如果伺服器正在接受新連🄵，則回傳 `True`。

在 3.7 版被加入.

async wait_closed()

等待*close()* 方法完成且所有活動連🄵都已結束。

sockets

伺服器正在監聽的類似 socket 的物件串列，`asyncio.trsock.TransportSocket`。

在 3.7 版的變更: 在 Python 3.7 之前，`Server.sockets` 曾經直接回傳🄵部伺服器 sockets 的串列。在 3.7 中回傳了該串列的副本。

事件🄵圈實作

asyncio 🄵附兩個不同的事件🄵圈實作：*SelectorEventLoop* 和*ProactorEventLoop*。

預設情🄵下，asyncio 被配置🄵要使用*EventLoop*。

class asyncio.**SelectorEventLoop**

基於*selectors* 模組的一個*AbstractEventLoop* 子類🄵。

使用特定平台上最有效的 *selector*。也可以手動配置要使用的確切 selector 實作：

```
import asyncio
import selectors

async def main():
    ...

loop_factory = lambda: asyncio.SelectorEventLoop(selectors.SelectSelector())
asyncio.run(main(), loop_factory=loop_factory)
```

可用性: Unix, Windows.

class asyncio.**ProactorEventLoop**

用於 Windows 的*AbstractEventLoop* 子類🄵，使用「I/O 完成埠 (IOCP, I/O Completion Ports)」。

可用性: Windows.

> **也參考**
>
> I/O 完成埠的 MSDN 文件。

class asyncio.**EventLoop**

An alias to the most efficient available subclass of *AbstractEventLoop* for the given platform.

在 Unix 上是*SelectorEventLoop* 的🄵名，在 Windows 上是*ProactorEventLoop* 的🄵名。

在 3.13 版被加入.

class asyncio.**AbstractEventLoop**

🄵符合 asyncio 標准的事件🄵圈的抽象基礎類🄵。

事件🄵圈方法 部分列出了替代 AbstractEventLoop 實作應該定義的所有方法。

范例

請注意，本節中的所有範例都 **故意**展示如何使用低階事件🄵圈 API，如*loop.run_forever()* 和*loop.call_soon()*。現代 asyncio 應用程式很少需要這種方式撰寫；請考慮使用高階的函式，如*asyncio.run()*。

使用 call_soon() 的 Hello World 范例

使用*loop.call_soon()* 方法排程回呼的範例。回呼會顯示 "Hello World"，然後停止事件🄵圈：

```
import asyncio

def hello_world(loop):
    """列印 'Hello World' 🄵停止事件🄵圈的回呼"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# 排程對 hello_world() 的呼叫
loop.call_soon(hello_world, loop)

# 阻塞呼叫被 loop.stop() 中斷
try:
    loop.run_forever()
finally:
    loop.close()
```

也參考

使用協程和*run()* 函式建立的類似*Hello World* 範例。

使用 call_later() 顯示目前日期

一個回呼的範例，每秒顯示目前日期。回呼使用*loop.call_later()* 方法在 5 秒後重新排程自己，然後停止事件🄴圈：

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# 排程 display_date() 的第一次呼叫
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# 阻塞呼叫被 loop.stop() 中斷
try:
    loop.run_forever()
finally:
    loop.close()
```

也參考

使用協程和*run()* 函式建立的類似*current date* 範例。

監聽檔案描述器以進行讀取事件

使用*loop.add_reader()* 方法等待檔案描述器接收到某些資料，然後關閉事件🄴圈：

```
import asyncio
from socket import socketpair

# 建立一對連接的檔案描述器
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # 我們完成了：🄵銷檔案描述器
    loop.remove_reader(rsock)

    # 停止事件🄴圈
    loop.stop()

# 🄵讀取事件🄵🄵檔案描述器
```

(繼續下一頁)

(繼續上一頁)

```
loop.add_reader(rsock, reader)

# 模擬從網路接收資料
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # 運行事件🄵圈
    loop.run_forever()
finally:
    # 我們完成了。關閉 socket 和事件🄵圈。
    rsock.close()
    wsock.close()
    loop.close()
```

也參考

- 使用傳輸、協定和*loop.create_connection()* 方法的類似範例。
- 另一個使用高階*asyncio.open_connection()* 函式和串流的類似範例。

設定 SIGINT 和 SIGTERM 的訊號處理程式

（此 `signals` 範例僅在 Unix 上運作。）

使用*loop.add_signal_handler()* 方法🄵🄵訊號 `SIGINT` 和 `SIGTERM` 的處理程式：

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

19.1.10 Futures

原始碼： Lib/asyncio/futures.py、source:‘Lib/asyncio/base_futures.py

---

Future 物件被用來連結**低階回呼式程式**和高階 async/await 程式。

Future 函式

asyncio.**isfuture**(*obj*)

如果 *obj* 🄵下面任意物件，回傳 `True`：

- 一個*asyncio.Future* 的實例、
- 一個*asyncio.Task* 的實例、
- 帶有 `_asyncio_future_blocking` 屬性的類 Future 物件 (Future-like object)。

在 3.5 版被加入.

asyncio.**ensure_future**(*obj*, *, *loop=None*)

回傳：

- *obj* 引數會保持原樣，*obj* 須🄵*Future*、*Task* 或類 Future 物件（可以用*isfuture()* 來進行檢查。）
- 包裝 (wrap) 了 *obj* 的*Task* 物件，如果 *obj* 是一個協程 (coroutine) (可以用*iscoroutine()* 來進行檢查)；在此情🄵下該協程將透過 `ensure_future()` 來排程。
- 一個會等待 *obj* 的*Task* 物件，*obj* 須🄵一個可等待物件（*inspect.isawaitable()* 用於測試。）

如果 *obj* 不是上述物件的話會引發一個*TypeError* 例外。

> **重要**
>
> 將參照 (reference) 儲存至此函式的結果，用以防止任務在執行中消失。
>
> 請見*create_task()* 函式，它是建立新 Task 的推薦方法。或者使用*asyncio.TaskGroup*，它會在🄵部保留對 Task 的參照。

在 3.5.1 版的變更: 這個函式接受任意*awaitable* 物件。

在 3.10 版之後被🄵用: 如果 *obj* 不是類 Future 物件且 *loop* 🄵未被指定，同時🄵有正在執行的事件🄵圈 (event loop)，則會發出🄵用警告。

asyncio.**wrap_future**(*future*, *, *loop=None*)

將一個*concurrent.futures.Future* 物件包裝到*asyncio.Future* 物件中。

在 3.10 版之後被🄵用: 如果 *future* 不是類 Future 物件且 *loop* 未被指定，同時🄵有正在執行的事件🄵圈，則會發出🄵用警告。

Future 物件

class asyncio.**Future**(*, *loop=None*)

一個 Future 代表一個非同步運算的最終結果。🄵不支援執行緒安全 (thread-safe)。

Future 是一個*awaitable* 物件。協程可以等待 Future 物件直到它們有結果或例外被設置、或者被取消。一個 Future 可被多次等待而結果都會是相同的。

Future 通常用於讓低階基於回呼的程式（例如在協定實作中使用 asyncio *transports*）能🄵與高階 async/await 程式互動。

經驗法則🄵永遠不要在提供給使用者的 API 中公開 Future 物件，同時建議使用*loop.create_future()* 來建立 Future 物件。如此一來，不同實作的事件🄵圈可以注入自己最佳化實作的 Future 物件。

在 3.7 版的變更: 加入對*contextvars* 模組的支援。

在 3.10 版之後被🄵用: 如果未指定 *loop* 🄵且🄵有正在執行的事件🄵圈則會發出🄵用警告。

result()

回傳 Future 的結果。

如果 Future 狀態🄵 *done*（完成），🄵擁有*set_result()* 方法設定的一個結果，則回傳該結果之值。

如果 Future 狀態🄵 *done*，🄵擁有*set_exception()* 方法設定的一個例外，那🄵這個方法會引發該例外。

如果 Future 已被 *cancelled*（取消），此方法會引發一個*CancelledError* 例外。

如果 Future 的結果還不可用，此方法會引發一個*InvalidStateError* 例外。

set_result(*result*)

將 Future 標記🄵 *done* 🄵設定其結果。

如果 Future 已經 *done* 則引發一個*InvalidStateError* 錯誤。

set_exception(*exception*)

將 Future 標記🄵 *done* 🄵設定一個例外。

如果 Future 已經 *done* 則引發一個*InvalidStateError* 錯誤。

done()

如果 Future 已🄵 *done* 則回傳 `True`。

如果 Future 有被 *cancelled*、*set_result()* 有被呼叫來🄵其設定結果、或*set_exception()* 有被呼叫🄵其設定例外，那🄵它就是 *done*。

cancelled()

如果 Future 已經被 *cancelled* 則回傳 `True`。

這個方法通常在🄵 Future 設定結果或例外前用來確認它還🄵被 *cancelled*：

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

新增一個在 Future 🄵 *done* 時執行的回呼函式。

呼叫 *callback* 🄵附帶做🄵唯一引數的 Future 物件。

如果呼叫這個方法時 Future 已經🄵 *done*，回呼函式會被*loop.call_soon()* 排程。

可選僅限關鍵字引數 *context* 用來指定一個讓 *callback* 執行於其中的客🄵化*contextvars.Context* 物件。如果🄵有提供 *context*，則使用目前情境。

可以用*functools.partial()* 傳遞引數給回呼函式，例如：

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

在 3.7 版的變更: 加入僅限關鍵字參數 *context*。更多細節請參🄵 **PEP 567**。

remove_done_callback(*callback*)

從回呼列表中移除 *callback*。

回傳被移除的回呼函式數量，通常🄵 1，除非一個回呼函式被多次加入。

cancel(*msg=None*)

取消 Future 🄵🄵回呼函式排程。

如果 Future 已經是 *done* 或 *cancelled*，回傳 `False`。否則將 Future 狀態改🄵 *cancelled* 🄵在🄵回呼函式排程後回傳 `True`。

在 3.9 版的變更: 新增 *msg* 參數。

exception()

回傳被設定於此 Future 的例外。

只有 Future 在 *done* 時才回傳例外（如果🄵有設定例外則回傳 None）。

如果 Future 已被 *cancelled*（取消），此方法會引發一個*CancelledError* 例外。

如果 Future 還不🄵 *done*，此方法會引發一個*InvalidStateError* 例外。

get_loop()

回傳已被 Future 物件🄵結 (bind) 的事件🄵圈。

在 3.7 版被加入.

這個例子建立一個 Future 物件，建立一個非同步 Task 🄵🄵其排程以設定 Future 結果，然後等待 Future 結果出現：

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

重要

該 Future 物件是🄵了模仿*concurrent.futures.Future* 而設計。主要差🄵包含：

- 與 asyncio 的 Future 不同，*concurrent.futures.Future* 實例不可被等待。
- *asyncio.Future.result()* 和*asyncio.Future.exception()* 不接受 *timeout* 引數。
- Future 不🄵 *done* 時*asyncio.Future.result()* 和*asyncio.Future.exception()* 會引發一個*InvalidStateError* 例外。
- 使用*asyncio.Future.add_done_callback()* 🄵🄵的回呼函式不會立即呼叫，而是被*loop.call_soon()* 排程。
- asyncio Future 不能與*concurrent.futures.wait()* 和*concurrent.futures.as_completed()* 函式相容。
- *asyncio.Future.cancel()* 接受一個可選的 msg 引數，但*concurrent.futures.Future.cancel()* 無此引數。

19.1.11 傳輸與協定

前言

Transports and Protocols are used by the **low-level** event loop APIs such as `loop.create_connection()`. They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both *Transports* and *Protocols*.

Introduction

At the highest level, the transport is concerned with *how* bytes are transmitted, while the protocol determines *which* bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport's point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

Most of connection oriented event loop methods (such as `loop.create_connection()`) usually accept a *protocol_factory* argument used to create a *Protocol* object for an accepted connection, represented by a *Transport* object. Such methods usually return a tuple of `(transport, protocol)`.

目🄵

This documentation page contains the following sections:

- The *Transports* section documents asyncio `BaseTransport`, `ReadTransport`, `WriteTransport`, `Transport`, `DatagramTransport`, and `SubprocessTransport` classes.
- The *Protocols* section documents asyncio `BaseProtocol`, `Protocol`, `BufferedProtocol`, `DatagramProtocol`, and `SubprocessProtocol` classes.
- The *Examples* section showcases how to work with transports, protocols, and low-level event loop APIs.

Transports

原始碼： Lib/asyncio/transports.py

Transports are classes provided by `asyncio` in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an *asyncio event loop*.

asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are *not thread safe*.

Transports Hierarchy

class asyncio.**BaseTransport**

Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.**WriteTransport**(*BaseTransport*)

A base transport for write-only connections.

Instances of the *WriteTransport* class are returned from the *loop.connect_write_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.**ReadTransport**(*BaseTransport*)

A base transport for read-only connections.

Instances of the *ReadTransport* class are returned from the *loop.connect_read_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the *Transport* class are returned from or used by event loop methods like *loop.create_connection()*, *loop.create_unix_connection()*, *loop.create_server()*, *loop.sendfile()*, etc.

class asyncio.**DatagramTransport**(*BaseTransport*)

A transport for datagram (UDP) connections.

Instances of the *DatagramTransport* class are returned from the *loop.create_datagram_endpoint()* event loop method.

class asyncio.**SubprocessTransport**(*BaseTransport*)

An abstraction to represent a connection between a parent and its child OS process.

Instances of the *SubprocessTransport* class are returned from event loop methods *loop.subprocess_shell()* and *loop.subprocess_exec()*.

Base Transport

BaseTransport.**close**()

Close the transport.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *protocol.connection_lost()* method will be called with *None* as its argument. The transport should not be used once it is closed.

BaseTransport.**is_closing**()

Return `True` if the transport is closing or is closed.

BaseTransport.**get_extra_info**(*name*, *default=None*)

Return information about the transport or underlying resources it uses.

name is a string representing the piece of transport-specific information to get.

default is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
  - `'peername'`: the remote address to which the socket is connected, result of *socket.socket.getpeername()* (`None` on error)

  - `'socket'`: *socket.socket* instance
  - `'sockname'`: the socket's own address, result of *socket.socket.getsockname()*
- SSL socket:
  - `'compression'`: the compression algorithm being used as a string, or `None` if the connection isn't compressed; result of *ssl.SSLSocket.compression()*
  - `'cipher'`: a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of *ssl.SSLSocket.cipher()*
  - `'peercert'`: peer certificate; result of *ssl.SSLSocket.getpeercert()*
  - `'sslcontext'`: *ssl.SSLContext* instance
  - `'ssl_object'`: *ssl.SSLObject* or *ssl.SSLSocket* instance
- pipe:
  - `'pipe'`: pipe object
- subprocess:
  - `'subprocess'`: *subprocess.Popen* instance

`BaseTransport.`**`set_protocol`**(*protocol*)

Set a new protocol.

Switching protocol should only be done when both protocols are documented to support the switch.

`BaseTransport.`**`get_protocol`**()

Return the current protocol.

Read-only Transports

`ReadTransport.`**`is_reading`**()

Return `True` if the transport is receiving new data.

在 3.7 版被加入.

`ReadTransport.`**`pause_reading`**()

Pause the receiving end of the transport. No data will be passed to the protocol's *protocol.data_received()* method until *resume_reading()* is called.

在 3.7 版的變更: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

`ReadTransport.`**`resume_reading`**()

Resume the receiving end. The protocol's *protocol.data_received()* method will be called once again if some data is available for reading.

在 3.7 版的變更: The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

`WriteTransport.`**`abort`**()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

`WriteTransport.`**`can_write_eof`**()

Return *True* if the transport supports *write_eof()*, *False* if not.

WriteTransport.**get_write_buffer_size**()

Return the current size of the output buffer used by the transport.

WriteTransport.**get_write_buffer_limits**()

Get the *high* and *low* watermarks for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use *set_write_buffer_limits()* to set the limits.

在 3.4.2 版被加入.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)

Set the *high* and *low* watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's *protocol.pause_writing()* and *protocol.resume_writing()* methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither *high* nor *low* can be negative.

pause_writing() is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, *resume_writing()* is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high watermark. Setting *high* to zero forces *low* to zero as well, and causes *pause_writing()* to be called whenever the buffer becomes non-empty. Setting *low* to zero causes *resume_writing()* to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

WriteTransport.**write**(*data*)

Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

WriteTransport.**writelines**(*list_of_data*)

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling *write()* on each element yielded by the iterable, but may be implemented more efficiently.

WriteTransport.**write_eof**()

Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closed connections.

Datagram Transports

DatagramTransport.**sendto**(*data*, *addr=None*)

Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is *None*, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

在 3.13 版的變更: This method can be called with an empty bytes object to send a zero-length datagram. The buffer size calculation used for flow control is also updated to account for the datagram header.

DatagramTransport.**abort**()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

Subprocess Transports

SubprocessTransport.**get_pid**()

Return the subprocess process id as an integer.

SubprocessTransport.**get_pipe_transport**(*fd*)

Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- `0`: writable streaming transport of the standard input (*stdin*), or *None* if the subprocess was not created with `stdin=PIPE`
- `1`: readable streaming transport of the standard output (*stdout*), or *None* if the subprocess was not created with `stdout=PIPE`
- `2`: readable streaming transport of the standard error (*stderr*), or *None* if the subprocess was not created with `stderr=PIPE`
- other *fd*: *None*

SubprocessTransport.**get_returncode**()

Return the subprocess return code as an integer or *None* if it hasn't returned, which is similar to the *subprocess.Popen.returncode* attribute.

SubprocessTransport.**kill**()

Kill the subprocess.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for *terminate()*.

另請參🄵*subprocess.Popen.kill()*。

SubprocessTransport.**send_signal**(*signal*)

Send the *signal* number to the subprocess, as in *subprocess.Popen.send_signal()*.

SubprocessTransport.**terminate**()

Stop the subprocess.

On POSIX systems, this method sends *SIGTERM* to the subprocess. On Windows, the Windows API function `TerminateProcess()` is called to stop the subprocess.

另請參🄵*subprocess.Popen.terminate()*。

SubprocessTransport.**close**()

Kill the subprocess by calling the *kill()* method.

If the subprocess hasn't returned yet, and close transports of *stdin*, *stdout*, and *stderr* pipes.

Protocols

原始碼： Lib/asyncio/protocols.py

---

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with *transports*.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Base Protocols

class asyncio.**BaseProtocol**

Base protocol with methods that all protocols share.

class asyncio.**Protocol**(*BaseProtocol*)

The base class for implementing streaming protocols (TCP, Unix sockets, etc).

class asyncio.**BufferedProtocol**(*BaseProtocol*)

A base class for implementing streaming protocols with manual control of the receive buffer.

class asyncio.**DatagramProtocol**(*BaseProtocol*)

The base class for implementing datagram (UDP) protocols.

class asyncio.**SubprocessProtocol**(*BaseProtocol*)

The base class for implementing protocols communicating with child processes (unidirectional pipes).

Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

Connection Callbacks

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

BaseProtocol.**connection_made**(*transport*)

Called when a connection is made.

The *transport* argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

BaseProtocol.**connection_lost**(*exc*)

Called when the connection is lost or closed.

The argument is either an exception object or *None*. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

Flow Control Callbacks

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the *set_write_buffer_limits()* method for more details.

BaseProtocol.**pause_writing**()

Called when the transport's buffer goes over the high watermark.

BaseProtocol.**resume_writing**()

Called when the transport's buffer drains below the low watermark.

If the buffer size equals the high watermark, *pause_writing()* is not called: the buffer size must go strictly over.

Conversely, *resume_writing()* is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

Streaming Protocols

Event methods, such as *loop.create_server()*, *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()*, and *loop.connect_write_pipe()* accept factories that return streaming protocols.

Protocol.**data_received**(*data*)

Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.

The method can be called an arbitrary number of times while a connection is open.

However, *protocol.eof_received()* is called at most once. Once `eof_received()` is called, `data_received()` is not called anymore.

`Protocol.`**`eof_received`**`()`

Called when the other end signals it won't send any more data (for example by calling *transport.write_eof()*, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don't support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Buffered Streaming Protocols

在 3.7 版被加入.

Buffered Protocols can be used with any event loop method that supports *Streaming Protocols*.

`BufferedProtocol` implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on *BufferedProtocol* instances:

`BufferedProtocol.`**`get_buffer`**`(`*sizehint*`)`

Called to allocate a new receive buffer.

sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what *sizehint* suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

`get_buffer()` must return an object implementing the buffer protocol.

`BufferedProtocol.`**`buffer_updated`**`(`*nbytes*`)`

Called when the buffer was updated with the received data.

nbytes is the total number of bytes that were written to the buffer.

`BufferedProtocol.`**`eof_received`**`()`

See the documentation of the *protocol.eof_received()* method.

get_buffer() can be called an arbitrary number of times during a connection. However, *protocol.eof_received()* is called at most once and, if called, *get_buffer()* and *buffer_updated()* won't be called after it.

State machine:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Datagram Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the *loop.create_datagram_endpoint()* method.

`DatagramProtocol.`**`datagram_received`**(*data*, *addr*)

Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

`DatagramProtocol.`**`error_received`**(*exc*)

Called when a previous send or receive operation raises an *OSError*. *exc* is the *OSError* instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

備🄵

On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears 'ready' and excess packets are dropped. An *OSError* with `errno` set to *errno.ENOBUFS* may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Subprocess Protocols

Subprocess Protocol instances should be constructed by protocol factories passed to the *loop.subprocess_exec()* and *loop.subprocess_shell()* methods.

`SubprocessProtocol.`**`pipe_data_received`**(*fd*, *data*)

Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

`SubprocessProtocol.`**`pipe_connection_lost`**(*fd*, *exc*)

Called when one of the pipes communicating with the child process is closed.

fd is the integer file descriptor that was closed.

`SubprocessProtocol.`**`process_exited`**()

Called when the child process has exited.

It can be called before *pipe_data_received()* and *pipe_connection_lost()* methods.

范例

TCP Echo Server

Create a TCP echo server using the *loop.create_server()* method, send back received data, and close the connection:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport
```

(繼續下一頁)

(繼續上一頁)

```
    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

也參考

The *TCP echo server using streams* example uses the high-level `asyncio.start_server()` function.

TCP Echo Client

A TCP echo client using the `loop.create_connection()` method, sends data, and waits until the connection is closed:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
```

(繼續下一頁)

(繼續上一頁)

```
    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

也參考

The *TCP echo client using streams* example uses the high-level `asyncio.open_connection()` function.

UDP Echo Server

A UDP echo server, using the `loop.create_datagram_endpoint()` method, sends back received data:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()
```

(繼續下一頁)

(繼續上一頁)

```
asyncio.run(main())
```

UDP Echo Client

A UDP echo client, using the *loop.create_datagram_endpoint()* method, sends data and closes the transport when it receives the answer:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the `loop.create_connection()` method with a protocol:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

也參考

The *watch a file descriptor for read events* example uses the low-level `loop.add_reader()` method to register an FD.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

loop.subprocess_exec() and SubprocessProtocol

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by the *loop.subprocess_exec()* method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False

    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # process_exited() method can be called before
        # pipe_connection_lost() method: wait until both methods are
        # called.
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using high-level APIs.

19.1.12 Policies

> **警告**
>
> Policies are deprecated and will be removed in Python 3.16. Users are encouraged to use the `asyncio.run()` function or the `asyncio.Runner` with *loop_factory* to use the desired loop implementation.

An event loop policy is a global object used to get and set the current *event loop*, as well as create new event loops. The default policy can be *replaced* with *built-in alternatives* to use different event loop implementations, or substituted by a *custom policy* that can override these behaviors.

The *policy object* gets and sets a separate event loop per *context*. This is per-thread by default, though custom policies could define *context* differently.

Custom event loop policies can control the behavior of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()`.

Policy objects should implement the APIs defined in the `AbstractEventLoopPolicy` abstract base class.

Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

`asyncio.`**`get_event_loop_policy`**`()`

Return the current process-wide policy.

在 3.14 版之後被🄵用: The `get_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

`asyncio.`**`set_event_loop_policy`**`(`*policy*`)`

Set the current process-wide policy to *policy*.

If *policy* is set to `None`, the default policy is restored.

在 3.14 版之後被🄵用: The `set_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

Policy Objects

The abstract event loop policy base class is defined as follows:

`class` `asyncio.`**`AbstractEventLoopPolicy`**

An abstract base class for asyncio policies.

`get_event_loop``()`

Get the event loop for the current context.

Return an event loop object implementing the `AbstractEventLoop` interface.

This method should never return `None`.

在 3.6 版的變更.

`set_event_loop``(`*loop*`)`

Set the event loop for the current context to *loop*.

`new_event_loop``()`

Create and return a new event loop object.

This method should never return `None`.

在 3.14 版之後被🄵用: The *AbstractEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

asyncio ships with the following built-in policies:

class asyncio.**DefaultEventLoopPolicy**

The default asyncio policy. Uses *SelectorEventLoop* on Unix and *ProactorEventLoop* on Windows.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

在 3.8 版的變更: On Windows, *ProactorEventLoop* is now used by default.

在 3.14 版的變更: The *get_event_loop()* method of the default asyncio policy now raises a *RuntimeError* if there is no set event loop.

在 3.14 版之後被🄵用: The *DefaultEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsSelectorEventLoopPolicy**

An alternative event loop policy that uses the *SelectorEventLoop* event loop implementation.

可用性: Windows.

在 3.14 版之後被🄵用: The *WindowsSelectorEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsProactorEventLoopPolicy**

An alternative event loop policy that uses the *ProactorEventLoop* event loop implementation.

可用性: Windows.

在 3.14 版之後被🄵用: The *WindowsProactorEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

Custom Policies

To implement a new event loop policy, it is recommended to subclass *DefaultEventLoopPolicy* and override the methods for which custom behavior is wanted, e.g.:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.13 平台支援

asyncio module（模組）被設計🄵可🄵的 (portable)，但由於平臺的底層架構和功能不同，在一些平臺上存在細微的差🄵和限制。

所有平台

- *loop.add_reader()* 和*loop.add_writer()* 不能用來監視檔案 I/O。

Windows

原 始 碼： Lib/asyncio/proactor_events.py、source:'Lib/asyncio/windows_events.py、source:'Lib/asyncio/windows_utils.py

---

在 3.8 版的變更: 在 Windows 上，現在*ProactorEventLoop* 是預設的事件🄴圈。

Windows 上的所有事件🄴圈都不支援以下 method（方法）：

- 不支援*loop.create_unix_connection()* 和*loop.create_unix_server()*。*socket.AF_UNIX* socket 系列常數僅限於 Unix 上使用。
- 不支援*loop.add_signal_handler()* 和*loop.remove_signal_handler()*。

SelectorEventLoop 有以下限制：

- *SelectSelector* 只被用於等待 socket 事件：它支援 socket 且最多支援 512 個 socket。
- *loop.add_reader()* 和*loop.add_writer()* 只接受 socket 處理函式（例如不支援 pipe 檔案描述器 (pipe file descriptor)）。
- 因🄵不支援 pipe，所以*loop.connect_read_pipe()* 和*loop.connect_write_pipe()* method 🄵有被實作出來。
- 不支援*子行程 (subprocess)*，也就是🄵*loop.subprocess_exec()* 和*loop.subprocess_shell()* method 🄵有被實作出來。

ProactorEventLoop 有以下限制：

- 不支援*loop.add_reader()* 和*loop.add_writer()* method。

Windows 上單調時鐘 (monotonic clock) 的解析度大約🄵 15.6 毫秒。最佳的解析度是 0.5 毫秒。解析度和硬體（HPET 是否可用）與 Windows 的設定有關。

Windows 的子行程支援

在 Windows 上，預設的事件🄴圈*ProactorEventLoop* 支援子行程，而*SelectorEventLoop* 則不支援。

macOS

完整支援現在普遍流行的 macOS 版本。

macOS <= 10.8

在 macOS 10.6、10.7 和 10.8 上，預設的事件🄴圈是使用*selectors.KqueueSelector*，在這些版本上它🄵不支援字元裝置 (character device)。可以手工設置*SelectorEventLoop* 來使用*SelectSelector* 或*PollSelector* 以在這些舊版 macOS 上支援字元裝置。例如：

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

19.1.14 擴充

The main direction for *asyncio* extending is writing custom *event loop* classes. Asyncio has helpers that could be used to simplify this task.

備🄵

Third-parties should reuse existing asyncio code with caution, a new Python version is free to break backward compatibility in *internal* part of API.

Writing a Custom Event Loop

asyncio.AbstractEventLoop declares very many methods. Implementing all them from scratch is a tedious job.

A loop can get many common methods implementation for free by inheriting from `asyncio.BaseEventLoop`.

In turn, the successor should implement a bunch of *private* methods declared but not implemented in `asyncio.BaseEventLoop`.

For example, `loop.create_connection()` checks arguments, resolves DNS addresses, and calls `loop._make_socket_transport()` that should be implemented by inherited class. The `_make_socket_transport()` method is not documented and is considered as an *internal* API.

Future and Task private constructors

asyncio.Future and *asyncio.Task* should be never created directly, please use corresponding *loop.create_future()* and *loop.create_task()*, or *asyncio.create_task()* factories instead.

However, third-party *event loops* may *reuse* built-in future and task implementations for the sake of getting a complex and highly optimized code for free.

For this purpose the following, *private* constructors are listed:

`Future.__init__`(*, *loop=None*)

Create a built-in future instance.

loop is an optional event loop instance.

`Task.__init__`(*coro*, *, *loop=None*, *name=None*, *context=None*)

Create a built-in task instance.

loop is an optional event loop instance. The rest of arguments are described in *loop.create_task()* description.

在 3.11 版的變更: *context* argument is added.

Task lifetime support

A third party task implementation should call the following functions to keep a task visible by *asyncio.all_tasks()* and *asyncio.current_task()*:

`asyncio._register_task`(*task*)

Register a new *task* as managed by *asyncio*.

Call the function from a task constructor.

`asyncio._unregister_task`(*task*)

Unregister a *task* from *asyncio* internal structures.

The function should be called when a task is about to finish.

`asyncio._enter_task`(*loop*, *task*)

Switch the current task to the *task* argument.

Call the function just before executing a portion of embedded *coroutine* (`coroutine.send()` or `coroutine.throw()`).

asyncio.**_leave_task**(*loop*, *task*)

Switch the current task back from *task* to `None`.

Call the function just after `coroutine.send()` or `coroutine.throw()` execution.

19.1.15 高階 API 索引

這個頁面列出了所有能使用 async/wait 的高階 asyncio API。

任務 (Tasks)

用於執行非同步程式、建立 Task 物件、帶有超時 (timeout) 設定地等待多個事件的工具程式。

run()	建立事件🄵圈 (event loop)、執行一個協程 (coroutine)、關閉事件🄵圈。
Runner	A context manager that simplifies multiple async function calls.
Task	Task 物件。
TaskGroup	A context manager that holds a group of tasks. Provides a convenient and reliable way to wait for all tasks in the group to finish.
create_task()	🄵動一個 asyncio 的 Task 物件，然後回傳它。
current_task()	回傳目前 Task 物件。
all_tasks()	回傳事件🄵圈中所有未完成的 task 物件。
await *sleep()*	休眠數秒鐘。
await *gather()*	🄵行 (concurrent) 地執行事件的排程與等待。
await *wait_for()*	有超時設置的執行。
await *shield()*	屏蔽取消操作。
await *wait()*	監控完成情🄵。
timeout()	Run with a timeout. Useful in cases when `wait_for` is not suitable.
to_thread()	在不同的 OS 執行緒 (thread) 中非同步地執行一個函式。
run_coroutine_threadsafe()	從其他 OS 執行緒中🄵一個協程排程。
for in *as_completed()*	用 `for` 🄵圈來監控完成情🄵。

范例

- 使用 *asyncio.gather()* 平行 *(parallel)* 執行。
- 使用 *asyncio.wait_for()* 🄵制設置超時。
- 取消任務。
- 使用 *asyncio.sleep()*。
- 請參🄵*Tasks* 文件頁面。

🄵列 (Queues)

🄵列應被用於🄵多個 asyncio Task 物件分配工作、實作 connection pools（連🄵池）以及 pub/sub（發行/訂🄵）模式。

Queue	一個先進先出 (FIFO) 🄵列。
PriorityQueue	一個優先🄵列 (priority queue)。
LifoQueue	一個後進先出 (LIFO) 🄵列。

范例

- 使用 *asyncio.Queue* 🄵多個 *Task* 分配工作。
- 請參🄵🄵列文件頁面。

子行程 (Subprocesses)

用於衍生（spawn）子行程和執行 shell 指令的工具程式。

`await create_subprocess_exec()`	建立一個子行程。
`await create_subprocess_shell()`	執行一個 shell 命令。

范例

- 執行一個 *shell* 指令。
- 請參🄵子行程 *APIs* 相關文件。

串流 (Streams)

用於網路 IO 處理的高階 API。

`await open_connection()`	建立一個 TCP 連🄵。
`await open_unix_connection()`	建立一個 Unix socket 連🄵。
`await start_server()`	🄵動一個 TCP 伺服器。
`await start_unix_server()`	🄵動一個 Unix socket 伺服器。
`StreamReader`	接收網路資料的高階 async/await 物件。
`StreamWriter`	傳送網路資料的高階 async/await 物件。

范例

- *TCP* 用🄵端範例。
- 請參🄵串流 *APIs* 文件。

同步化 (Synchronization)

類似執行緒且能被用於 Task 中的同步化原始物件 (primitive)。

`Lock`	一個互斥鎖 (mutex lock)。
`Event`	一個事件物件。
`Condition`	一個條件物件。
`Semaphore`	一個旗號 (semaphore) 物件。
`BoundedSemaphore`	一個有界的旗號物件。
`Barrier`	一個屏障物件。

范例

- 使用 *asyncio.Event*。
- 使用 *asyncio.Barrier*。
- 請參🄵 asyncio 關於同步化原始物件的文件。

例外

asyncio.CancelledError	當一 Task 物件被取消時被引發。請參🄵*Task.cancel()*。
asyncio.BrokenBarrierError	當一 Barrier 物件損壞時被引發。請參🄵*Barrier.wait()*。

范例

- *在取消請求發生的程式中處理 CancelledError 例外*。
- 請參🄵*asyncio 專用例外*完整列表。

19.1.16 低階 API 索引

本頁列出所有低階 asyncio APIs。

取得事件🄵圈

asyncio.get_running_loop()	**推薦使用**於取得目前運行事件🄴圈 (event loop) 的函式。
asyncio.get_event_loop()	獲得一個（正在運行的或透過目前 policy 建立的）事件🄵圈實例。
asyncio.set_event_loop()	透過目前 policy 來設定目前事件🄵圈。
asyncio.new_event_loop()	建立一個新的事件🄵圈。

范例

- *使用 asyncio.get_running_loop()*。

事件🄵圈方法

也請查🄵文件中關於*事件🄵圈方法*的主要段落。

生命🄵期

loop.run_until_complete()	執行一個 Future/Task/awaitable（可等待物件）直到完成。
loop.run_forever()	持續運行事件🄵圈。
loop.stop()	停止事件🄵圈。
loop.close()	關閉事件🄵圈。
loop.is_running()	如果事件🄵圈正在執行則回傳 `True`。
loop.is_closed()	如果事件🄵圈已經被關閉則回傳 `True`。
`await` *loop.shutdown_asyncgens()*	關閉非同步🄵生器 (asynchronous generators)。

除錯

loop.set_debug()	開🄴或禁用除錯模式。
loop.get_debug()	取得目前除錯模式。

🄵回呼函式排程

loop.call_soon()	🄵快呼叫回呼函式 (callback)。
loop.call_soon_threadsafe()	*loop.call_soon()* 方法之有支援執行緒安全 (thread-safe) 變體。
loop.call_later()	在給定時間之後呼叫回呼函式。
loop.call_at()	在給定時間當下呼叫回呼函式。

執行緒 (Thread)/直譯器 (Interpreter)/行程池 (Process Pool)

await *loop.run_in_executor()*	在*concurrent.futures* 執行器 (excutor) 中執行一個 CPU 密集型 (CPU-bound) 或其它阻塞型式的函式。
loop.set_default_executor()	🄵*loop.run_in_executor()* 設定預設執行器。

Tasks 與 Futures

loop.create_future()	建立一個*Future* 物件。
loop.create_task()	像是*Task* 一樣，🄵協程 (coroutine) 排程。
loop.set_task_factory()	設定被*loop.create_task()* 用來建立*Tasks* 的工廠函式 (factory)。
loop.get_task_factory()	取得被*loop.create_task()* 用來建立*Tasks* 的工廠函式。

DNS

await *loop.getaddrinfo()*	非同步版本的*socket.getaddrinfo()*。
await *loop.getnameinfo()*	非同步版本的*socket.getnameinfo()*。

網路和 IPC

await *loop.create_connection()*	開🄵一個 TCP 連🄵。
await *loop.create_server()*	建立一個 TCP 伺服器。
await *loop.create_unix_connection()*	開🄵一個 Unix socket 連🄵。
await *loop.create_unix_server()*	建立一個 Unix socket 伺服器。
await *loop.connect_accepted_socket()*	將*socket* 包裝成 `(transport, protocol)`。
await *loop.create_datagram_endpoint()*	開🄵一個資料單元 (datagram) (UDP) 連🄵。
await *loop.sendfile()*	透過傳輸通道傳送一個檔案。
await *loop.start_tls()*	將一個已存在的連🄵升級到 TLS。
await *loop.connect_read_pipe()*	將 pipe（管道）讀取端包裝成 `(transport, protocol)`。
await *loop.connect_write_pipe()*	將 pipe 寫入端包裝成 `(transport, protocol)`。

Sockets

await *loop.sock_recv()*	從*socket* 接收資料。
await *loop.sock_recv_into()*	將從*socket* 接收到的資料存放於一個緩衝區 (buffer) 中。
await *loop.sock_recvfrom()*	從*socket* 接收一個資料單元。
await *loop.sock_recvfrom_into()*	將從*socket* 接收到的資料單元存放於一個緩衝區中。
await *loop.sock_sendall()*	傳送資料到*socket*。
await *loop.sock_sendto()*	透過*socket* 將資料單元傳送至給定的地址。
await *loop.sock_connect()*	連接*socket*。
await *loop.sock_accept()*	接受一個*socket* 連🄵。
await *loop.sock_sendfile()*	透過*socket* 傳送一個檔案。
loop.add_reader()	開始監控一個檔案描述器 (file descriptor) 的可讀取性。
loop.remove_reader()	停止監控一個檔案描述器的可讀取性。
loop.add_writer()	開始監控一個檔案描述器的可寫入性。
loop.remove_writer()	停止監控一個檔案描述器的可寫入性。

Unix 信號

loop.add_signal_handler()	🄵*signal* 新增一個處理函式 (handler)。
loop.remove_signal_handler()	🄵除*signal* 的處理函式。

子行程

loop.subprocess_exec()	衍生 (spawn) 一個子行程 (subprocess)。
loop.subprocess_shell()	從 shell 指令衍生一個子行程。

錯誤處理

loop.call_exception_handler()	呼叫例外處理函式。
loop.set_exception_handler()	設定一個新的例外處理函式。
loop.get_exception_handler()	取得目前例外處理函式。
loop.default_exception_handler()	預設例外處理函式實作。

范例

- 使用 *asyncio.new_event_loop()* 和 *loop.run_forever()*。
- 使用 *loop.call_later()*。
- 使用 `loop.create_connection()` 以實作一個 *echo* 用🄵端。
- 使用 `loop.create_connection()` 來連接 *socket*。
- 使用 *add_reader()* 監控 *FD* 的讀取事件。
- 使用 *loop.add_signal_handler()*。
- 使用 *loop.add_signal_handler()*。

傳輸

所有傳輸方式都有實作以下方法：

transport.close()	關閉傳輸。
transport.is_closing()	如果傳輸正在關閉或已經關閉則回傳 `True`。
transport.get_extra_info()	請求傳輸的相關資訊。
transport.set_protocol()	設定一個新協定。
transport.get_protocol()	回傳目前協定。

可以接收資料（TCP 和 Unix 連🄵、pipe 等）的傳輸。它由*loop.create_connection()*、*loop.create_unix_connection()*、*loop.connect_read_pipe()* 等方法回傳：

讀取傳輸

transport.is_reading()	如果傳輸正在接收則回傳 `True`。
transport.pause_reading()	暫停接收。
transport.resume_reading()	繼續接收。

可以傳送資料（TCP 和 Unix 連🄵、pipe 等）的傳輸。它由*loop.create_connection()*、*loop.create_unix_connection()*、*loop.connect_write_pipe()* 等方法回傳：

寫入傳輸

transport.write()	將資料寫入傳輸。
transport.writelines()	將緩衝區資料寫入傳輸。
transport.can_write_eof()	如果傳輸支援傳送 EOF 則回傳*True*。
transport.write_eof()	在清除 (flush) 已緩衝的資料後關閉傳輸🄵傳送 EOF。
transport.abort()	立即關閉傳輸。
transport.get_write_buffer_size()	回傳目前輸出緩衝區的大小。
transport.get_write_buffer_limits()	回傳用於寫入流量控制 (write flow control) 的高低標記位 (high and low water marks)。
transport.set_write_buffer_limits()	🄵寫入流量控制設定高低標記位。

由*loop.create_datagram_endpoint()* 回傳的傳輸：

資料單元傳輸

transport.sendto()	傳送資料到連🄵遠端。
transport.abort()	立即關閉傳輸。

基於子行程的低階傳輸抽象，它會由*loop.subprocess_exec()* 和*loop.subprocess_shell()* 所回傳：

子行程傳輸

transport.get_pid()	回傳子行程的行程 id。
transport.get_pipe_transport()	回傳被請求用於通訊 pipe （*stdin*、*stdout* 或 *stderr*）的傳輸。
transport.get_returncode()	回傳子行程的回傳代號 (return code)。
transport.kill()	殺死子行程。
transport.send_signal()	傳送一個訊號到子行程。
transport.terminate()	停止子行程。
transport.close()	殺死子行程🄵關閉所有 pipes。

協定

協定類🄵可以實作以下**回呼方法**:

callback *connection_made()*	在連🄵建立時被呼叫。
callback *connection_lost()*	在失去連🄵或連🄵關閉時被呼叫。
callback *pause_writing()*	在傳輸緩衝區超過高標記位時被呼叫。
callback *resume_writing()*	在傳輸緩衝區低於低標記位時被呼叫。

串流協定 (TCP, Unix socket, Pipes)

callback *data_received()*	在接收到資料時被呼叫。
callback *eof_received()*	在接收到 EOF 時被呼叫。

緩沖串流協定

callback *get_buffer()*	呼叫後會分配新的接收緩衝區。
callback *buffer_updated()*	在以接收到的資料更新緩衝區時被呼叫。
callback *eof_received()*	在接收到 EOF 時被呼叫。

資料單元協定

callback *datagram_received()*	在接收到資料單元時被呼叫。
callback *error_received()*	在前一個傳送或接收操作引發*OSError* 時被呼叫。

子行程協定

callback *pipe_data_received()*	在子行程向 *stdout* 或 *stderr* pipe 寫入資料時被呼叫。
callback *pipe_connection_lost()*	在與子行程通訊的其中一個 pipes 關閉時被呼叫。
callback *process_exited()*	在子行程退出時呼叫。它可以在*pipe_data_received()* 和*pipe_connection_lost()* 方法之前呼叫。

事件🄵圈 Policies

Policy 是改變*asyncio.get_event_loop()* 這類函式行🄵的一個低階機制。更多細節請見*Policy 相關段落*。

存取 Policy

asyncio.get_event_loop_policy()	回傳目前整個行程中的 Policy。
asyncio.set_event_loop_policy()	設定整個行程中的一個新 Policy。
AbstractEventLoopPolicy	Policy 物件的基礎類🄵。

19.1.17 使用 asyncio 開發

非同步程式設計 (asynchronous programming) 與傳統的" 順序" 程式設計 (sequential programming) 不同。

本頁列出常見的錯誤和陷阱，🄵解釋如何避免它們。

除錯模式

在預設情🄵下 asyncio 以正式生🄵模式 (production mode) 執行。🄵了讓開發更輕松，asyncio 還有一種除錯模式 *(debug mode)*。

有幾種方法可以🄵用 asyncio 除錯模式：

- 將 `PYTHONASYNCIODEBUG` 環境變數設定🄵 `1`。
- 使用*Python 開發模式 (Development Mode)*。
- 將 `debug=True` 傳遞給*`asyncio.run()`*。
- 呼叫*`loop.set_debug()`*。

除了🄵用除錯模式外，還要考慮：

- 將*asyncio logger*（日🄵記🄵器）的日🄵級別設置🄵*`logging.DEBUG`*，例如下面的程式片段可以在應用程式🄵動時運行：

```
logging.basicConfig(level=logging.DEBUG)
```

- 配置*`warnings`* 模組以顯示*`ResourceWarning`* 警告。一種方法是使用 `-W default` 命令列選項。

🄵用除錯模式時：

- 許多非執行緒安全 (non-threadsafe) 的 asyncio APIs（例如*`loop.call_soon()`* 和*`loop.call_at()`* 方法），如果從錯誤的執行緒呼叫就會引發例外。
- 如果執行一個 I/O 操作花費的時間太長，則將 I/O 選擇器 (selector) 的執行時間記🄵到日🄵中。
- 執行時間超過 100 毫秒的回呼 (callback) 將會被記🄵於日🄵。屬性*`loop.slow_callback_duration`* 可用於設定以秒🄵單位的最小執行持續時間，超過這個值執行時間就會被視🄵" 緩慢"。

🄵行性和多執行緒 (Concurrency and Multithreading)

事件🄵圈在執行緒中運行（通常是主執行緒），🄵在其執行緒中執行所有回呼和 Tasks（任務）。當一個 Task 在事件🄵圈中運行時，🄵有其他 Task 可以在同一個執行緒中運行。當一個 Task 執行一個 `await` 運算式時，正在執行的 Task 會被暫停，而事件🄵圈會執行下一個 Task。

要從不同的 OS 執行緒🄵一個*callback* 排程，應該使用*`loop.call_soon_threadsafe()`* 方法。例如：

```
loop.call_soon_threadsafe(callback, *args)
```

幾乎所有 asyncio 物件都不支援執行緒安全 (thread safe)，這通常不是問題，除非在 Task 或回呼函式之外有程式需要和它們一起運作。如果需要這樣的程式來呼叫低階 asyncio API，應該使用*`loop.call_soon_threadsafe()`* 方法，例如：

```
loop.call_soon_threadsafe(fut.cancel)
```

要從不同的 OS 執行緒🄵一個協程物件排程，應該使用*`run_coroutine_threadsafe()`* 函式。它會回傳一個*`concurrent.futures.Future`* 以存取結果：

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

🄵了能🄵處理訊號，事件🄵圈必須於主執行緒中運行。

`loop.run_in_executor()` 方法可以和*`concurrent.futures.ThreadPoolExecutor`* 或*`InterpreterPoolExecutor`* 一起使用，這能🄵在作業系統上另一個不同的執行緒中執行阻塞程式，且避免阻塞執行事件🄵圈的執行緒。

目前🄵有什🄵辦法能直接從另一個行程（例如透過*multiprocessing* 🄵動的程序）來🄵協程或回呼排程。事件🄵圈方法小節列出了可以從 pipes（管道）讀取🄵監視 file descriptor（檔案描述器）而不會阻塞事件🄵圈的 API。此外，asyncio 的子行程 API 提供了一種🄵動行程🄵從事件🄵圈與其通訊的辦法。最後，之前提到的*loop.run_in_executor()* 方法也可和*concurrent.futures.ProcessPoolExecutor* 搭配使用，以在另一個行程中執行程式。

執行阻塞的程式

不應該直接呼叫阻塞（CPU 密集型）程式。例如一個執行 1 秒 CPU 密集型計算的函式，那🄵所有🄵行非同步 Tasks 和 IO 操作都會被延遲 1 秒。

一個 executor（執行器）可以被用來在不同的執行緒（包含不同直譯器）、或甚至不同的行程中執行任務，以避免事件🄵圈阻塞了 OS 執行緒。詳情請見*loop.run_in_executor()* 方法。

日🄵記🄵

asyncio 使用*logging* 模組，所有日🄵記🄵都是透過 `"asyncio"` logger 執行的。

日🄵級🄵被預設🄵*logging.INFO*，它可以很容易地被調整：

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

網路日🄵記🄵可能會阻塞事件🄵圈。建議使用獨立的執行緒來處理日🄵或使用非阻塞 IO，範例請參見 blocking-handlers。

偵測從未被等待的 (never-awaited) 協程

當協程函式被呼叫而不是被等待時（即執行 `coro()` 而不是 `await coro()`）或者協程🄵有透過*asyncio.create_task()* 被排程，asyncio 將會發出*RuntimeWarning*：

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

輸出：

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

除錯模式中的輸出：

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

常用的修復方法是去等待協程或者呼叫*asyncio.create_task()* 函式：

```
async def main():
    await test()
```

偵測從未被取得的 (never-retrieved) 例外

如果呼叫*Future.set_exception()*，但 Future 物件從未被等待，例外將無法被傳播 (propagate) 到使用者程式。在這種情🄵下，當 Future 物件被垃圾回收 (garbage collected) 時，asyncio 將發出一則日🄵訊息。

未處理例外的例子：

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

輸出：

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

🄵用除錯模式以取得任務建立處的追🄵資訊 (traceback)：

```
asyncio.run(main(), debug=True)
```

除錯模式中的輸出：

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

備🄵

asyncio 的原始碼可以在 Lib/asyncio/ 中找到。

19.2 socket —低階網路介面

原始碼： Lib/socket.py

這個模組提供了操作 BSD *socket* 的介面。這在所有現代 Unix 系統、Windows、MacOS，以及一些其他平台上都可用。

備🄵

由於是呼叫作業系統的 socket API，某些行🄵可能會因平台而有所差🄵。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵 *WebAssembly 平台*。

Python 的介面是將 Unix 的系統呼叫和 socket 函式庫介面直接轉🄵成 Python 的物件導向風格：`socket()` 函式會回傳一個 *socket* 物件，這個物件的方法實作了各種 socket 系統呼叫。與 C 語言介面相比，參數型🄵較🄵高階：就像 Python 文件操作中的 `read()` 和 `write()` 一樣，接收操作時會自動分配緩衝區，而發送操作時的緩衝區長度則是隱含的。

也參考

***socketserver* 模組**
簡化編寫網路伺服器的類🄴。

***ssl* 模組**
對 socket 物件的 TLS/SSL 的包裝器 (wrapper)。

19.2.1 Socket 系列家族

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an *AF_UNIX* socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  在 3.3 版的變更: Previously, *AF_UNIX* socket paths were assumed to use UTF-8 encoding.

  在 3.5 版的變更: Writable *bytes-like object* is now accepted.

- A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.

  - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

- For *AF_INET6* address family, a four-tuple `(host, port, flowinfo, scope_id)` is used, where *flowinfo* and *scope_id* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For *socket* module methods, *flowinfo* and *scope_id* can be omitted just for backward compatibility. Note, however, omission of *scope_id* can cause problems in manipulating scoped IPv6 addresses.

  在 3.7 版的變更: For multicast addresses (with *scope_id* meaningful) *address* may not contain `%scope_id` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).

- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.

- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a

tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:

  - *addr_type* is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
  - *scope* is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
  - If *addr_type* is `TIPC_ADDR_NAME`, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is `TIPC_ADDR_NAMESEQ`, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is `TIPC_ADDR_ID`, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.

- A tuple `(interface, )` is used for the *AF_CAN* address family, where *interface* is a string representing a network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.
  - *CAN_ISOTP* protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
  - *CAN_J1939* protocol require a tuple `(interface, name, pgn, addr)` where additional parameters are 64-bit unsigned integer representing the ECU name, a 32-bit unsigned integer representing the Parameter Group Number (PGN), and an 8-bit integer representing the address.
- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  在 3.3 版被加入.
- *AF_BLUETOOTH* supports the following protocols and address formats:
  - *BTPROTO_L2CAP* accepts a tuple `(bdaddr, psm[, cid[, bdaddr_type]])` where:
    - `bdaddr` is a string specifying the Bluetooth address.
    - `psm` is an integer specifying the Protocol/Service Multiplexer.
    - `cid` is an optional integer specifying the Channel Identifier. If not given, defaults to zero.
    - `bdaddr_type` is an optional integer specifying the address type; one of *BDADDR_BREDR* (default), *BDADDR_LE_PUBLIC*, *BDADDR_LE_RANDOM*.

    在 3.14 版的變更: Added `cid` and `bdaddr_type` fields.
  - *BTPROTO_RFCOMM* accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
  - *BTPROTO_HCI* accepts a format that depends on your OS.
    - On Linux it accepts an integer `device_id` or a tuple `(device_id, [channel])` where `device_id` specifies the number of the Bluetooth device, and `channel` is an optional integer specifying the HCI channel (*HCI_CHANNEL_RAW* by default).
    - On FreeBSD, NetBSD and DragonFly BSD it accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string.

    在 3.2 版的變更: 加入對 NetBSD 和 DragonFlyBSD 的支援。

    在 3.13.3 版的變更: 新增對 FreeBSD 的支援。

    在 3.14 版的變更: Added `channel` field. `device_id` not packed in a tuple is now accepted.
  - *BTPROTO_SCO* accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string or a *bytes* object. (ex. `'12:23:34:45:56:67'` or `b'12:23:34:45:56:67'`)

    在 3.14 版的變更: 新增對 FreeBSD 的支援。
- *AF_ALG* is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:

- *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
- *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
- *feat* and *mask* are unsigned 32bit integers.

可用性: Linux >= 2.6.38.

Some algorithm types require more recent Kernels.

在 3.6 版被加入.

- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

  可用性: Linux >= 3.9

  請見 *vsock(7)*

  在 3.7 版被加入.

- *AF_PACKET* is a low-level interface directly to network devices. The addresses are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
  - *ifname* - String specifying the device name.
  - *proto* - The Ethernet protocol number. May be *ETH_P_ALL* to capture all protocols, one of the *ETHERTYPE_* constants* or any other Ethernet protocol number.
  - *pkttype* - Optional integer specifying the packet type:
    * `PACKET_HOST` (the default) - Packet addressed to the local host.
    * `PACKET_BROADCAST` - Physical-layer broadcast packet.
    * `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
    * `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
    * `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
  - *hatype* - Optional integer specifying the ARP hardware address type.
  - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

  可用性: Linux >= 2.2.

- *AF_QIPCRTR* is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

  可用性: Linux >= 4.7.

  在 3.8 版被加入.

- `IPPROTO_UDPLITE` is a variant of UDP which allows you to specify what portion of a packet is covered with the checksum. It adds two socket options that you can change. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` will change what portion of outgoing packets are covered by the checksum and `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` will filter out packets which cover too little of their data. In both cases `length` should be in `range(8, 2**16, 8)`.

  Such a socket should be constructed with `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv4 or `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv6.

  可用性: Linux >= 2.6.20, FreeBSD >= 10.1

  在 3.9 版被加入.

- *AF_HYPERV* is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a `(vm_id, service_id)` tuple where the `vm_id` and `service_id` are UUID strings.

  The `vm_id` is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on `socket` are:

  - `HV_GUID_ZERO`
  - `HV_GUID_BROADCAST`
  - `HV_GUID_WILDCARD` - Used to bind on itself and accept connections from all partitions.
  - `HV_GUID_CHILDREN` - Used to bind on itself and accept connection from child partitions.
  - `HV_GUID_LOOPBACK` - Used as a target to itself.
  - `HV_GUID_PARENT` - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

  The `service_id` is the service identifier of the registered service.

  在 3.12 版被加入.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise *OSError* or one of its subclasses.

Non-blocking mode is supported through *setblocking()*. A generalization of this based on timeouts is supported through *settimeout()*.

19.2.2 模組🄵容

The module *socket* exports the following elements.

例外

exception `socket.`**error**

一個已🄵用的*OSError* 的🄵名。

在 3.3 版的變更: Following **PEP 3151**, this class was made an alias of *OSError*.

exception `socket.`**herror**

A subclass of *OSError*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *gethostbyname_ex()* and *gethostbyaddr()*. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

在 3.3 版的變更: This class was made a subclass of *OSError*.

exception `socket.`**gaierror**

A subclass of *OSError*, this exception is raised for address-related errors by *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

在 3.3 版的變更: This class was made a subclass of *OSError*.

exception `socket.`**timeout**

A deprecated alias of *TimeoutError*.

A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always "timed out".

在 3.3 版的變更: This class was made a subclass of *OSError*.

在 3.10 版的變更: This class was made an alias of *TimeoutError*.

常數

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` *IntEnum* collections.

在 3.4 版被加入.

socket.**AF_UNIX**

socket.**AF_INET**

socket.**AF_INET6**

These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**AF_UNSPEC**

AF_UNSPEC means that *getaddrinfo()* should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

socket.**SOCK_RAW**

socket.**SOCK_RDM**

socket.**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.**SOCK_CLOEXEC**

socket.**SOCK_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

> **也參考**
>
> Secure File Descriptor Handling for a more thorough explanation.

可用性: Linux >= 2.6.27.

在 3.2 版被加入.

SO_*

socket.**SOMAXCONN**

MSG_*

SOL_*

SCM_*

IPPROTO_*

IPPORT_*

INADDR_*

IP_*

IPV6_*

EAI_*

AI_*

NI_*

TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the *setsockopt()* and *getsockopt()* methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

在 3.6 版的變更: SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, SO_PASSSEC, TCP_USER_TIMEOUT, TCP_CONGESTION were added.

在 3.6.5 版的變更: Added support for TCP_FASTOPEN, TCP_KEEPCNT on Windows platforms when available.

在 3.7 版的變更: 新增 TCP_NOTSENT_LOWAT。

Added support for TCP_KEEPIDLE, TCP_KEEPINTVL on Windows platforms when available.

在 3.10 版的變更: IP_RECVTOS was added. Added TCP_KEEPALIVE. On MacOS this constant can be used in the same way that TCP_KEEPIDLE is used on Linux.

在 3.11 版的變更: Added TCP_CONNECTION_INFO. On MacOS this constant can be used in the same way that TCP_INFO is used on Linux and BSD.

在 3.12 版的變更: Added SO_RTABLE and SO_USER_COOKIE. On OpenBSD and FreeBSD respectively those constants can be used in the same way that SO_MARK is used on Linux. Also added missing TCP socket options from Linux: TCP_MD5SIG, TCP_THIN_LINEAR_TIMEOUTS, TCP_THIN_DUPACK, TCP_REPAIR, TCP_REPAIR_QUEUE, TCP_QUEUE_SEQ, TCP_REPAIR_OPTIONS, TCP_TIMESTAMP, TCP_CC_INFO, TCP_SAVE_SYN, TCP_SAVED_SYN, TCP_REPAIR_WINDOW, TCP_FASTOPEN_CONNECT, TCP_ULP, TCP_MD5SIG_EXT, TCP_FASTOPEN_KEY, TCP_FASTOPEN_NO_COOKIE, TCP_ZEROCOPY_RECEIVE, TCP_INQ, TCP_TX_DELAY. Added IP_PKTINFO, IP_UNBLOCK_SOURCE, IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP.

在 3.13 版的變更: Added SO_BINDTOIFINDEX. On Linux this constant can be used in the same way that SO_BINDTODEVICE is used, but with the index of a network interface instead of its name.

在 3.14 版的變更: Added missing IP_FREEBIND, IP_RECVERR, IPV6_RECVERR, IP_RECVTTL, and IP_RECVORIGDSTADDR on Linux.

在 3.14 版的變更: Added support for TCP_QUICKACK on Windows platforms when available.

socket.**AF_CAN**

socket.**PF_CAN**

SOL_CAN_*

CAN_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

可用性: Linux >= 2.6.25, NetBSD >= 8.

在 3.3 版被加入.

在 3.11 版的變更: NetBSD support was added.

在 3.14 版的變更: Restored missing CAN_RAW_ERR_FILTER on Linux.

socket.**CAN_BCM**

CAN_BCM_*

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

可用性: Linux >= 2.6.25.

> **備🄵**
>
> The `CAN_BCM_CAN_FD_FRAME` flag is only available on Linux >= 4.8.

在 3.4 版被加入.

`socket.`**`CAN_RAW_FD_FRAMES`**

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

可用性: Linux >= 3.6.

在 3.5 版被加入.

`socket.`**`CAN_RAW_JOIN_FILTERS`**

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

This constant is documented in the Linux documentation.

可用性: Linux >= 4.1.

在 3.9 版被加入.

`socket.`**`CAN_ISOTP`**

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

可用性: Linux >= 2.6.25.

在 3.7 版被加入.

`socket.`**`CAN_J1939`**

CAN_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

可用性: Linux >= 5.4.

在 3.9 版被加入.

`socket.`**`AF_DIVERT`**

`socket.`**`PF_DIVERT`**

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

可用性: FreeBSD >= 14.0.

在 3.12 版被加入.

`socket.`**`AF_PACKET`**

`socket.`**`PF_PACKET`**

`PACKET_*`

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

可用性: Linux >= 2.2.

`socket.`**`ETH_P_ALL`**

`ETH_P_ALL` can be used in the *socket* constructor as *proto* for the *AF_PACKET* family in order to capture every packet, regardless of protocol.

For more information, see the *packet(7)* manpage.

可用性: Linux.

在 3.12 版被加入.

socket.**AF_RDS**

socket.**PF_RDS**

socket.**SOL_RDS**

RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

可用性: Linux >= 2.6.30.

在 3.3 版被加入.

socket.**SIO_RCVALL**

socket.**SIO_KEEPALIVE_VALS**

socket.**SIO_LOOPBACK_FAST_PATH**

RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as arguments to the *ioctl()* method of socket objects.

在 3.6 版的變更: 加入 SIO_LOOPBACK_FAST_PATH。

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.**AF_ALG**

socket.**SOL_ALG**

ALG_*

Constants for Linux Kernel cryptography.

可用性: Linux >= 2.6.38.

在 3.6 版被加入.

socket.**AF_VSOCK**

socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**

VMADDR*

SO_VM*

Constants for Linux host/guest communication.

可用性: Linux >= 4.8.

在 3.7 版被加入.

socket.**AF_LINK**

可用性: BSD, macOS.

在 3.4 版被加入.

socket.**has_ipv6**

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.**AF_BLUETOOTH**

socket.**BTPROTO_L2CAP**

socket.**BTPROTO_RFCOMM**

socket.**BTPROTO_HCI**

socket.**BTPROTO_SCO**

Integer constants for use with Bluetooth addresses.

socket.**BDADDR_ANY**

socket.**BDADDR_LOCAL**

These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with *BTPROTO_RFCOMM*.

socket.**BDADDR_BREDR**

socket.**BDADDR_LE_PUBLIC**

socket.**BDADDR_LE_RANDOM**

These constants describe the Bluetooth address type when binding or connecting a *BTPROTO_L2CAP* socket.

可用性: Linux, FreeBSD

在 3.14 版被加入.

socket.**SOL_RFCOMM**

socket.**SOL_L2CAP**

socket.**SOL_HCI**

socket.**SOL_SCO**

socket.**SOL_BLUETOOTH**

Used in the level argument to the *setsockopt()* and *getsockopt()* methods of Bluetooth socket objects.

SOL_BLUETOOTH is only available on Linux. Other constants are available if the corresponding protocol is supported.

SO_L2CAP_*

socket.**L2CAP_LM**

L2CAP_LM_*

SO_RFCOMM_*

RFCOMM_LM_*

SO_SCO_*

SO_BTH_*

BT_*

Used in the option name and value argument to the *setsockopt()* and *getsockopt()* methods of Bluetooth socket objects.

`BT_*` and *L2CAP_LM* are only available on Linux. `SO_BTH_*` are only available on Windows. Other constants may be available on Linux and various BSD platforms.

在 3.14 版被加入.

socket.**HCI_FILTER**

socket.**HCI_TIME_STAMP**

socket.**HCI_DATA_DIR**

socket.**SO_HCI_EVT_FILTER**

socket.**SO_HCI_PKT_FILTER**

Option names for use with *BTPROTO_HCI*. Availability and format of the option values depend on platform.

在 3.14 版的變更: Added `SO_HCI_EVT_FILTER` and `SO_HCI_PKT_FILTER` on NetBSD and DragonFly BSD. Added `HCI_DATA_DIR` on FreeBSD, NetBSD and DragonFly BSD.

socket.**HCI_DEV_NONE**

The `device_id` value used to create an HCI socket that isn't specific to a single Bluetooth adapter.

可用性: Linux

在 3.14 版被加入.

socket.**HCI_CHANNEL_RAW**

socket.**HCI_CHANNEL_USER**

socket.**HCI_CHANNEL_MONITOR**

socket.**HCI_CHANNEL_CONTROL**

socket.**HCI_CHANNEL_LOGGING**

Possible values for `channel` field in the *BTPROTO_HCI* address.

可用性: Linux

在 3.14 版被加入.

socket.**AF_QIPCRTR**

Constant for Qualcomm's IPC router protocol, used to communicate with service providing remote processors.

可用性: Linux >= 4.7.

socket.**SCM_CREDS2**

socket.**LOCAL_CREDS**

socket.**LOCAL_CREDS_PERSISTENT**

LOCAL_CREDS and LOCAL_CREDS_PERSISTENT can be used with SOCK_DGRAM, SOCK_STREAM sockets, equivalent to Linux/DragonFlyBSD SO_PASSCRED, while LOCAL_CREDS sends the credentials at first read, LOCAL_CREDS_PERSISTENT sends for each read, SCM_CREDS2 must be then used for the latter for the message type.

在 3.11 版被加入.

可用性: FreeBSD.

socket.**SO_INCOMING_CPU**

Constant to optimize CPU locality, to be used in conjunction with `SO_REUSEPORT`.

在 3.11 版被加入.

可用性: Linux >= 3.9

socket.**SO_REUSEPORT_LB**

Constant to enable duplicate address and port bindings with load balancing.

在 3.14 版被加入.

可用性: FreeBSD >= 12.0

socket.**AF_HYPERV**

socket.**HV_PROTOCOL_RAW**

socket.**HVSOCKET_CONNECT_TIMEOUT**

socket.**HVSOCKET_CONNECT_TIMEOUT_MAX**

socket.**HVSOCKET_CONNECTED_SUSPEND**

socket.**HVSOCKET_ADDRESS_FLAG_PASSTHRU**

socket.**HV_GUID_ZERO**

socket.**HV_GUID_WILDCARD**

socket.**HV_GUID_BROADCAST**

socket.**HV_GUID_CHILDREN**

socket.**HV_GUID_LOOPBACK**

socket.**HV_GUID_PARENT**

Constants for Windows Hyper-V sockets for host/guest communications.

可用性: Windows.

在 3.12 版被加入.

socket.**ETHERTYPE_ARP**

socket.**ETHERTYPE_IP**

socket.**ETHERTYPE_IPV6**

socket.**ETHERTYPE_VLAN**

IEEE 802.3 protocol number. constants.

可用性: Linux, FreeBSD, macOS.

在 3.12 版被加入.

socket.**SHUT_RD**

socket.**SHUT_WR**

socket.**SHUT_RDWR**

These constants are used by the *shutdown()* method of socket objects.

可用性: not WASI.

函式

建立 sockets

The following functions all create *socket objects*.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of CAN_RAW, *CAN_BCM*, *CAN_ISOTP* or *CAN_J1939*.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

The newly created socket is *non-inheritable*.

引發一個附帶引數 `self`、`family`、`type`、`protocol` 的稽核事件 `socket.__new__`。

在 3.3 版的變更: The AF_CAN family was added. The AF_RDS family was added.

在 3.4 版的變更: 新增 CAN_BCM 協定。

在 3.4 版的變更: The returned socket is now non-inheritable.

在 3.7 版的變更: 新增 CAN_ISOTP 協定。

在 3.7 版的變更: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system `socket()` call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support SOCK_NONBLOCK, but `sock.type` will be set to `socket.SOCK_STREAM`.

在 3.9 版的變更: 新增 CAN_J1939 協定。

在 3.10 版的變更: 新增 IPPROTO_MPTCP 協定。

socket.**socketpair**([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the *socket()* function above. The default family is *AF_UNIX* if defined on the platform; otherwise, the default is *AF_INET*.

The newly created sockets are *non-inheritable*.

在 3.2 版的變更: The returned socket objects now support the whole socket API, rather than a subset.

在 3.4 版的變更: The returned sockets are now non-inheritable.

在 3.5 版的變更: 新增對 Windows 的支援。

socket.**create_connection**(*address*, *timeout=GLOBAL_DEFAULT*, *source_address=None*, *, *all_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than *socket.connect()*: if *host* is a non-numeric hostname, it will try to resolve it for both *AF_INET* and *AF_INET6*, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by *getdefaulttimeout()* is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are ” or 0 respectively the OS default behavior will be used.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all_errors* is `True`, it is an *ExceptionGroup* containing the errors of all attempts.

在 3.2 版的變更: 新增 *source_address*。

在 3.11 版的變更: 新增 *all_errors*。

socket.**create_server**(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and returns the socket object.

family should be either *AF_INET* or *AF_INET6*. *backlog* is the queue size passed to *socket.listen()*; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true, *family* is *AF_INET6* and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise *ValueError*. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by *socket.getpeername()* when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with *has_dualstack_ipv6()*:

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

備🄵

On POSIX platforms the `SO_REUSEADDR` socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME_WAIT state.

在 3.8 版被加入.

socket.**has_dualstack_ipv6**()

Return `True` if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

在 3.8 版被加入.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object's *fileno()* method) and build a socket object from the result. Address family, socket type and protocol number are as for the *socket()* function above. The file descriptor should refer to a socket, but this is not checked --- subsequent operations on the

object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is *non-inheritable*.

在 3.4 版的變更: The returned socket is now non-inheritable.

`socket.`**`fromshare`**(*data*)

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

可用性: Windows.

在 3.3 版被加入.

`socket.`**`SocketType`**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

其他函式

The `socket` module also offers various network-related services:

`socket.`**`close`**(*fd*)

Close a socket file descriptor. This is like `os.close()`, but for sockets. On some platforms (most noticeable Windows) `os.close()` does not work for socket file descriptors.

在 3.7 版被加入.

`socket.`**`getaddrinfo`**(*host*, *port*, *family=AF_UNSPEC*, *type=0*, *proto=0*, *flags=0*)

This function wraps the C function `getaddrinfo` of the underlying system.

Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to provide options and limit the list of addresses returned. Pass their default values (`AF_UNSPEC`, 0, and 0, respectively) to not limit the results. See the note below for details.

The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the `socket()` function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for `AF_INET`, a `(address, port, flowinfo, scope_id)` 4-tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

> **備🄵**
>
> If you intend to use results from `getaddrinfo()` to create a socket (rather than, for example, retrieve *canonname*), consider limiting the results by *type* (e.g. `SOCK_STREAM` or `SOCK_DGRAM`) and/or *proto* (e.g. `IPPROTO_TCP` or `IPPROTO_UDP`) that your application can handle.
>
> The behavior with default values of *family*, *type*, *proto* and *flags* is system-specific.

Many systems (for example, most Linux configurations) will return a sorted list of all matching addresses. These addresses should generally be tried in order until a connection succeeds (possibly tried in parallel, for example, using a Happy Eyeballs algorithm). In these cases, limiting the *type* and/or *proto* can help eliminate unsuccessful or unusable connection attempts.

Some systems will, however, only return a single address. (For example, this was reported on Solaris and AIX configurations.) On these systems, limiting the *type* and/or *proto* helps ensure that this address is usable.

引發一個附帶引數 `host`、`port`、`family`、`type`、`protocol` 的稽核事件 `socket.getaddrinfo`。

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (socket.AF_INET, socket.SOCK_STREAM,
 6, '', ('93.184.216.34', 80))]
```

在 3.2 版的變更: parameters can now be passed using keyword arguments.

在 3.7 版的變更: for IPv6 multicast addresses, string representing an address will not contain `%scope_id` part.

`socket.`**`getfqdn`**`(`[*name*]`)`

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by *gethostbyaddr()* is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from *gethostname()* is returned.

`socket.`**`gethostbyname`**`(`*hostname*`)`

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See *gethostbyname_ex()* for a more complete interface. *gethostbyname()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

引發一個附帶引數 `hostname` 的稽核事件 `socket.gethostbyname`。

可用性: not WASI.

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host's primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). *gethostbyname_ex()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

引發一個附帶引數 `hostname` 的稽核事件 `socket.gethostbyname`。

可用性: not WASI.

`socket.`**`gethostname`**`()`

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

引發一個不附帶引數的稽核事件 `socket.gethostname`。

Note: *gethostname()* doesn't always return the fully qualified domain name; use *getfqdn()* for that.

可用性: not WASI.

socket.**gethostbyaddr**(*ip_address*)

Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function *getfqdn()*. *gethostbyaddr()* supports both IPv4 and IPv6.

引發一個附帶引數 `ip_address` 的稽核事件 `socket.gethostbyaddr`。

可用性: not WASI.

socket.**getnameinfo**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope_id` is appended to the host part if *sockaddr* contains meaningful *scope_id*. Usually this happens for multicast addresses.

For more information about *flags* you can consult *getnameinfo(3)*.

引發一個附帶引數 `sockaddr` 的稽核事件 `socket.getnameinfo`。

可用性: not WASI.

socket.**getprotobyname**(*protocolname*)

Translate an internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the *socket()* function. This is usually only needed for sockets opened in "raw" mode (*SOCK_RAW*); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

可用性: not WASI.

socket.**getservbyname**(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

引發一個附帶引數 `sockaddr`、`protocolname` 的稽核事件 `socket.getservbyname`。

可用性: not WASI.

socket.**getservbyport**(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

引發一個附帶引數 `port`、`protocolname` 的稽核事件 `socket.getservbyport`。

可用性: not WASI.

socket.**ntohl**(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**ntohs**(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

在 3.10 版的變更: Raises *OverflowError* if *x* does not fit in a 16-bit unsigned integer.

socket.**htonl**(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**htons**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

在 3.10 版的變更: Raises *OverflowError* if *x* does not fit in a 16-bit unsigned integer.

socket.**inet_aton**(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page *inet(3)* for details.

If the IPv4 address string passed to this function is invalid, *OSError* will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

inet_aton() does not support IPv6, and *inet_pton()* should be used instead for IPv4/v6 dual stack support.

socket.**inet_ntoa**(*packed_ip*)

Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, *OSError* will be raised. *inet_ntoa()* does not support IPv6, and *inet_ntop()* should be used instead for IPv4/v6 dual stack support.

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

socket.**inet_pton**(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. *inet_pton()* is useful when a library or network protocol calls for an object of type `in_addr` (similar to *inet_aton()*) or `in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the IP address string *ip_string* is invalid, *OSError* will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

可用性: Unix, Windows.

在 3.4 版的變更: Windows support added

socket.**inet_ntop**(*address_family*, *packed_ip*)

Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). *inet_ntop()* is useful when a library or network protocol returns an object of type `in_addr` (similar to *inet_ntoa()*) or `in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the bytes object *packed_ip* is not the correct length for the specified address family, *ValueError* will be raised. *OSError* is raised for errors from the call to *inet_ntop()*.

可用性: Unix, Windows.

在 3.4 版的變更: Windows support added

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

socket.**CMSG_LEN**(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for *recvmsg()* to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use *CMSG_SPACE()* and thus include space for padding, even when the item will be the last in the buffer. Raises *OverflowError* if *length* is outside the permissible range of values.

可用性: Unix, not WASI.

Most Unix platforms.

在 3.3 版被加入.

socket.**CMSG_SPACE**(*length*)

Return the buffer size needed for *recvmsg()* to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the *CMSG_SPACE()* values for their associated data lengths. Raises *OverflowError* if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

可用性: Unix, not WASI.

most Unix platforms.

在 3.3 版被加入.

socket.**getdefaulttimeout**()

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

socket.**setdefaulttimeout**(*timeout*)

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See *settimeout()* for possible values and their respective meanings.

socket.**sethostname**(*name*)

Set the machine's hostname to *name*. This will raise an *OSError* if you don't have enough rights.

引發一個附帶引數 `name` 的稽核事件 `socket.sethostname`。

可用性: Unix, not Android.

在 3.3 版被加入.

socket.**if_nameindex**()

Return a list of network interface information (index int, name string) tuples. *OSError* if the system call fails.

可用性: Unix, Windows, not WASI.

在 3.3 版被加入.

在 3.8 版的變更: 增加對 Windows 的支援。

備🄵

On Windows network interfaces have different names in different contexts (all names are examples):

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- name: `ethernet_32770`
- friendly name: `vEthernet (nat)`
- description: `Hyper-V Virtual Ethernet Adapter`

This function returns names of the second form from the list, `ethernet_32770` in this example case.

socket.**if_nametoindex**(*if_name*)

Return a network interface index number corresponding to an interface name. *OSError* if no interface with the given name exists.

可用性: Unix, Windows, not WASI.

在 3.3 版被加入.

在 3.8 版的變更: 增加對 Windows 的支援。

也參考

”Interface name” is a name as documented in `if_nameindex()`.

`socket.`**`if_indextoname`**(*if_index*)

Return a network interface name corresponding to an interface index number. `OSError` if no interface with the given index exists.

可用性: Unix, Windows, not WASI.

在 3.3 版被加入.

在 3.8 版的變更: 增加對 Windows 的支援。

也參考

”Interface name” is a name as documented in `if_nameindex()`.

`socket.`**`send_fds`**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an `AF_UNIX` socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult `sendmsg()` for the documentation of these parameters.

可用性: Unix, not WASI.

Unix platforms supporting `sendmsg()` and `SCM_RIGHTS` mechanism.

在 3.9 版被加入.

`socket.`**`recv_fds`**(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an `AF_UNIX` socket *sock*. Return `(msg, list(fds), flags, addr)`. Consult `recvmsg()` for the documentation of these parameters.

可用性: Unix, not WASI.

Unix platforms supporting `recvmsg()` and `SCM_RIGHTS` mechanism.

在 3.9 版被加入.

備🄵

Any truncated integers at the end of the list of file descriptors.

19.2.3 Socket 物件

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

在 3.2 版的變更: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `close()`.

`socket.`**`accept`**()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

The newly created socket is *non-inheritable*.

在 3.4 版的變更: The socket is now non-inheritable.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**bind**(*address*)

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family --- see above.)

引發一個附帶引數 `self`、`address` 的稽核事件 `socket.bind`。

可用性: not WASI.

socket.**close**()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from *makefile()* are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly, or to use a `with` statement around them.

在 3.6 版的變更: *OSError* is now raised if an error occurs when the underlying `close()` call is made.

> **備🄵**
>
> *close()* releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call *shutdown()* before *close()*.

socket.**connect**(*address*)

Connect to a remote socket at *address*. (The format of *address* depends on the address family --- see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a *TimeoutError* on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an *InterruptedError* exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

引發一個附帶引數 `self`、`address` 的稽核事件 `socket.connect`。

在 3.5 版的變更: The method now waits until the connection completes instead of raising an *InterruptedError* exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** for the rationale).

可用性: not WASI.

socket.**connect_ex**(*address*)

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

引發一個附帶引數 `self`、`address` 的稽核事件 `socket.connect`。

可用性: not WASI.

socket.**detach**()

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

在 3.2 版被加入.

socket.**dup**()

Duplicate the socket.

The newly created socket is *non-inheritable*.

在 3.4 版的變更: The socket is now non-inheritable.

可用性: not WASI.

socket.**fileno**()

Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with *select.select()*.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as *os.fdopen()*). Unix does not have this limitation.

socket.**get_inheritable**()

Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

在 3.4 版被加入.

socket.**getpeername**()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family --- see above.) On some systems this function is not supported.

socket.**getsockname**()

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family --- see above.)

socket.**getsockopt**(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (*SO_** *etc.*) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as byte strings).

可用性: not WASI.

socket.**getblocking**()

Return `True` if socket is in blocking mode, `False` if in non-blocking.

這等同於檢查 `socket.gettimeout() != 0`。

在 3.7 版被加入.

socket.**gettimeout**()

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to *setblocking()* or *settimeout()*.

socket.**ioctl**(*control*, *option*)

The *ioctl()* method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic *fcntl.fcntl()* and *fcntl.ioctl()* functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

可用性: Windows

在 3.6 版的變更: 加入 `SIO_LOOPBACK_FAST_PATH`。

socket.**listen**([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

可用性: not WASI.

在 3.5 版的變更: The *backlog* parameter is now optional.

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to `makefile()`. These arguments are interpreted the same way as by the built-in `open()` function, except the only supported *mode* values are `'r'` (default), `'w'`, `'b'`, or a combination of those.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by `makefile()` won't close the original socket unless all other file objects have been closed and `socket.close()` has been called on the socket object.

> **備🄵**
>
> On Windows, the file-like object created by `makefile()` cannot be used where a file object with a file descriptor is expected, such as the stream arguments of `subprocess.Popen()`.

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client has disconnected. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

socket.**recvfrom**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair `(bytes, address)` where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family --- see above.)

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

在 3.7 版的變更: For multicast IPv6 address, first item of *address* does not contain `%scope_id` part anymore. In order to get full IPv6 address use `getnameinfo()`.

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a `bytes` object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a `bytes` object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a `bytes` object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, *recvmsg()* will issue a *RuntimeWarning*, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the SCM_RIGHTS mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also *sendmsg()*.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

可用性: Unix.

Most Unix platforms.

在 3.3 版被加入.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as *recvmsg()* would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. *bytearray* objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (*sysconf()* value SC_IOV_MAX) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for *recvmsg()*.

The return value is a 4-tuple: (nbytes, ancdata, msg_flags, address), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for *recvmsg()*.

範例:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

可用性: Unix.

Most Unix platforms.

在 3.3 版被加入.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is

a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family --- see above.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the socket-howto.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()*, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

在 3.5 版的變更: The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for *recv()* above. Return the number of bytes sent. (The format of *address* depends on the address family --- see above.)

引發一個附帶引數 `self`、`address` 的稽核事件 `socket.sendto`。

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. *bytes* objects); the operating system may set a limit (*sysconf()* value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without *CMSG_SPACE()*) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for *send()*. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an *AF_UNIX* socket, on systems which support the `SCM_RIGHTS` mechanism. See also *recvmsg()*.

```
import socket, array
```

(繼續下一頁)

(繼續上一頁)

```
def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.array("i",␣
↪fds))])
```

可用性: Unix, not WASI.

Most Unix platforms.

引發一個附帶引數 `self`、`address` 的稽核事件 `socket.sendmsg`。

在 3.3 版被加入.

在 3.5 版的變更: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendmsg_afalg**([*msg*,]*, *op*[, *iv*[, *assoclen*[, *flags*]]])

Specialized version of *sendmsg()* for *AF_ALG* socket. Set mode, IV, AEAD associated data length and flags for *AF_ALG* socket.

可用性: Linux >= 2.6.38.

在 3.6 版被加入.

socket.**sendfile**(*file*, *offset=0*, *count=None*)

Send a file until EOF is reached by using high-performance *os.sendfile* and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If *os.sendfile* is not available (e.g. Windows) or *file* is not a regular file *send()* will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case *file.tell()* can be used to figure out the number of bytes which were sent. The socket must be of *SOCK_STREAM* type. Non-blocking sockets are not supported.

在 3.5 版被加入.

socket.**set_inheritable**(*inheritable*)

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

在 3.4 版被加入.

socket.**setblocking**(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *settimeout()* calls:

- `sock.setblocking(True)` 等價於 `sock.settimeout(None)`
- `sock.setblocking(False)` 等價於 `sock.settimeout(0.0)`

在 3.7 版的變更: The method no longer applies *SOCK_NONBLOCK* flag on *socket.type*.

socket.**settimeout**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

在 3.7 版的變更: The method no longer toggles *SOCK_NONBLOCK* flag on *socket.type*.

socket.**setsockopt**(*level*, *optname*, *value: int*)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in this module (SO_* etc. <socket-unix-constants>). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When *value* is set to `None`, *optlen* argument is required. It's equivalent to call `setsockopt()` C function with `optval=NULL` and `optlen=optlen`.

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

在 3.6 版的變更: setsockopt(level, optname, None, optlen: int) form added.

可用性: not WASI.

socket.**shutdown**(*how*)

Shut down one or both halves of the connection. If *how* is *SHUT_RD*, further receives are disallowed. If *how* is *SHUT_WR*, further sends are disallowed. If *how* is *SHUT_RDWR*, further sends and receives are disallowed.

可用性: not WASI.

socket.**share**(*process_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using *fromshare()*. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

可用性: Windows.

在 3.3 版被加入.

Note that there are no methods `read()` or `write()`; use *recv()* and *send()* without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the *socket* constructor.

socket.**family**

The socket family.

socket.**type**

The socket type.

socket.**proto**

The socket protocol.

19.2.4 Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling *setdefaulttimeout()*.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the *select* module can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a *timeout* exception) or if the system returns an error.

備🄵

At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the *fileno()* of a socket.

Timeouts and the `connect` method

The *connect()* operation is also subject to the timeout setting, and in general it is recommended to call *settimeout()* before calling *connect()* or pass a timeout parameter to *create_connection()*. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If *getdefaulttimeout()* is not *None*, sockets returned by the *accept()* method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by *accept()* is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by *accept()* is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

19.2.5 范例

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence *socket()*, *bind()*, *listen()*, *accept()* (possibly repeating the *accept()* to service more than one client), while a client only needs the sequence *socket()*, *connect()*. Also note that the server does not *sendall()*/*recv()* on the socket it is listening on but on the new socket returned by *accept()*.

前兩個範例只支援 IPv4：

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (*`CAN_BCM`*) the socket, you can use the *`socket.send()`* and *`socket.recv()`* operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
```

(繼續下一頁)

(繼續上一頁)

```
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

也參考

For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

19.3 `ssl` --- socket 物件的 TLS/SSL 包裝器

原始碼： Lib/ssl.py

這個模組向用🄴端及伺服器端提供了對於網路 socket 的傳輸層安全性協定（或稱🄴「安全通訊協定 (Secure Sockets Layer)」）加密及身分驗證功能。這個模組使用 OpenSSL 套件，它可以在所有的 Unix 系統、Windows、macOS、以及其他任何可能的平台上使用，只要事先在該平台上安裝 OpenSSL。

備🄵

由於呼叫了作業系統的 socket APIs，有些行🄴會根據平台而有所不同。OpenSSL 的安裝版本也會對模組的運作🄵生影響。例如，OpenSSL 版本 1.1.1 附帶 TLSv1.3。

警告

在使用此模組之前，請🄴讀*Security considerations*。如果不這樣做，可能會🄵生錯誤的安全性認知，因🄵 ssl 模組的預設設定未必適合你的應用程式。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

這個章節記🄵了 `ssl` 模組的物件及函式；關於 TSL、SSL、以及憑證的更多資訊，可以去參考此章節底部的「詳情」部分。

此模組提供了一個*ssl.SSLSocket* 類🄵，它是從*socket.socket* 衍生出來的，🄵且提供類似 socket 的包裝器，讓使用 SSL 進行資料傳輸時，可以進行資料的加密及解密。它也提供了一些額外的方法，如 `getpeercert()`，用於取得連結另一端的憑證；以及 `cipher()`，用於搜尋用於安全連接的加密方法 (cipher)；和 `get_verified_chain()`、`get_unverified_chain()` 能用於取得憑證鏈。

對於更🄴雜的應用程式，*ssl.SSLContext* 類🄴有助於管理設定及認證，然後可以透過*SSLContext.wrap_socket()* 方法建立的 SSL socket 繼承這些設定和認證。

在 3.5.3 版的變更: 更新以支援與 OpenSSL 1.1.0 進行連結

在 3.6 版的變更: OpenSSL 0.9.8, 1.0.0 及 1.0.1 版本已被🄴用且不再支援。在未來 ssl 模組將需要至少 OpenSSL 1.0.2 版本或 1.1.0 版本。

在 3.10 版的變更: **PEP 644** 已經被實作。ssl 模組需要 OpenSSL 1.1.1 以上的版本才能使用。

使用已經被🄴用的常數或函式將會導致🄵用警示。

19.3.1 函式、常數與例外

Socket 建立

SSLSocket 實例必須使用*SSLContext.wrap_socket()* 方法來建立。輔助函式*create_default_context()* 會回傳有安全預設設定的新語境 (context)。

使用預設語境及 IPv4/IPv6 雙協定堆🄵的用🄵端 socket 範例：

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

使用自訂語境及 IPv4 的用🄵端 socket 範例：

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

在本地 IPv4 上監聽伺服器 socket 的範例：

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

語境建立

一個可以幫忙建立出*SSLContext* 物件以用於一般目的的方便函式。

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *, *cafile=None*, *capath=None*, *cadata=None*)

回傳一個新的*SSLContext* 物件，使用給定 *purpose* 的預設值。這些設定是由*ssl* 選擇，通常比直接呼叫*SSLContext* 有更高的安全性。

cafile、*capath*、*cadata* 是用來選擇用於憑證認證的 CA 憑證，就像*SSLContext.load_verify_locations()* 一樣。如果三個值都是*None*，此函式會自動選擇系統預設的 CA 憑證。

這些設定包含：*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER*、*OP_NO_SSLv2*、以及*OP_NO_SSLv3*，使用高等加密套件但不包含 RC4 和未經身份驗證的加密套件。如果將 *purpose* 設定🄵*SERVER_AUTH*，則會把*verify_mode* 設🄵*CERT_REQUIRED* 🄵使用設定的 CA 憑證 (當 *cafile*、*capath* 或 *cadata* 其中一個值有被設定時) 或使用預設的 CA 憑證*SSLContext.load_default_certs()* 。

當系統有支援*keylog_filename* 🄵且有設定環境變數 SSLKEYLOGFILE 時*create_default_context()* 會🄵用密鑰日🄵記🄵 (logging)。

The default settings for this context include *VERIFY_X509_PARTIAL_CHAIN* and *VERIFY_X509_STRICT*. These make the underlying OpenSSL implementation behave more like a conforming implementation of **RFC 5280**, in exchange for a small amount of incompatibility with older X.509 certificates.

備🄵

協定、選項、加密方式和其它設定可以在不舍🄵舊值的情🄵下直接更改成新的值，這些值代表了在相容性和安全性之間取得的合理平衡。

如果你的應用程式需要特殊的設定，你應該要自行建立一個*SSLContext* 🄵自行調整設定。

備🄵

如果你發現某些舊的用🄵端或伺服器常適用此函式建立的*SSLContext* 連🄵時，收到"Protocol or cipher suite mismatch" 錯誤，這可能是因🄵他們的系統僅支援 SSL3.0，然而 SSL3.0 已被此函式用*OP_NO_SSLv3* 排除。目前廣泛認🄵 SSL3.0 已經被完全破解。如果你仍然希望在允許 SSL3.0 連🄵的情🄵下使用此函式，可以使用下面的方法：

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

備🄵

This context enables *VERIFY_X509_STRICT* by default, which may reject pre-**RFC 5280** or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using:

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

在 3.4 版被加入.

在 3.4.4 版的變更: 把 RC4 從預設加密方法字串中舍🄵。

在 3.6 版的變更: 把 ChaCha20/Poly1305 加入預設加密方法字串。

把 3DES 從預設加密方法字串中舍🄵。

在 3.8 版的變更: 增加了 SSLKEYLOGFILE 對密鑰日🄵記🄵 (logging) 的支援。

在 3.10 版的變更: 目前語境使用*PROTOCOL_TLS_CLIENT* 協定或*PROTOCOL_TLS_SERVER* 協定而非通用的*PROTOCOL_TLS*。

在 3.13 版的變更: The context now uses *VERIFY_X509_PARTIAL_CHAIN* and *VERIFY_X509_STRICT* in its default verify flags.

例外

exception ssl.**SSLError**

引發由底層 SSL 實作（目前由 OpenSSL 函式庫提供）所引發的錯誤訊息。這表示在覆蓋底層網路連🄵的高階加密和身份驗證層中存在一些問題。這項錯誤是*OSError* 的一個子型🄵。*SSLError* 實例的錯誤程式代碼和訊息是由 OpenSSL 函式庫提供。

在 3.3 版的變更: *SSLError* 曾經是*socket.error* 的一個子型🄵。

library

一個字串符號 (string mnemonic)，用來指定發生錯誤的 OpenSSL 子模組，如：SSL、PEM 或 X509。可能值的範圍取🄵於 OpenSSL 的版本。

在 3.3 版被加入.

reason

一個字串符號，用來指定發生錯誤的原因，如：CERTIFICATE_VERIFY_FAILED。可能值的範圍取🄵於 OpenSSL 的版本。

在 3.3 版被加入.

exception ssl.**SSLZeroReturnError**

一個*SSLError* 的子類🄵，當嘗試去讀寫已經被完全關閉的 SSL 連🄵時會被引發。請注意，這🄵不表示底層傳輸（例如 TCP）已經被關閉。

在 3.3 版被加入.

exception ssl.**SSLWantReadError**

一個*SSLError* 的子類🄵，當嘗試去讀寫資料前，底層 TCP 傳輸需要先接收更多資料時會由非阻塞的 *SSL socket* 引發該錯誤。

在 3.3 版被加入.

exception ssl.**SSLWantWriteError**

一個*SSLError* 的子類🄵，當嘗試去讀寫資料前，底層 TCP 傳輸需要先發送更多資料時會由非阻塞的 *SSL socket* 引發該錯誤。

在 3.3 版被加入.

exception ssl.**SSLSyscallError**

一個*SSLError* 的子類🄵，當嘗試去操作 SSL socket 時有系統錯誤🄵生會引發此錯誤。不幸的是，目前🄵有任何簡單的方法可以去檢查原本的的 errno 編號。

在 3.3 版被加入.

exception ssl.**SSLEOFError**

一個*SSLError* 的子類🄵，當 SSL 連🄵被突然終止時會引發此錯誤。通常，當此錯誤發生時，你不該再去重新使用底層傳輸。

在 3.3 版被加入.

exception ssl.**SSLCertVerificationError**

當憑證驗證失敗時會引發的一個*SSLError* 子類🄵。

在 3.7 版被加入.

verify_code

一個表示驗證錯誤的錯誤數值編號。

verify_message

一個人類可讀的驗證錯誤字串。

exception ssl.**CertificateError**

SSLCertVerificationError 的🄵名。

在 3.7 版的變更: 此例外現在是*SSLCertVerificationError* 的🄵名。

隨機🄵生

ssl.**RAND_bytes**(*num*, /)

回傳 *num* 個加密性🄵的🄵隨機位元組。如果 PRNG 未使用足🄵的資料做🄵隨機種子 (seed) 或是目前的 RAND 方法不支持該操作則會導致*SSLError* 錯誤。*RAND_status()* 函式可以用來檢查 PRNG 函式，而*RAND_add()* 則可以用來🄵 PRNG 設定隨機種子。

在幾乎所有的應用程式中，*os.urandom()* 會是較好的選擇。

請🄵讀維基百科的密碼學安全🄵隨機數🄵生器 (CSPRNG)文章來了解密碼學安全🄵隨機數🄵生器的需求。

在 3.3 版被加入.

ssl.**RAND_status**()

如果 SSL 🄵隨機數🄵生器已經使用「足🄵的」隨機性進行隨機種子生成，則回傳 `True`，否則回傳 `False`。你可以使用 `ssl.RAND_egd()` 函式和*ssl.RAND_add()* 函式來增加🄵隨機數🄵生器的隨機性。

ssl.**RAND_add**(*bytes*, *entropy*, /)

將給定的 *bytes* 混進 SSL 隨機🄵隨機數🄵生器中。*entropy* 參數（float 值）是指字串中包含熵值的下限（因此你可以將其設🄵 `0.0`）。請參🄵 **RFC 1750** 了解有關熵源的更多資訊。

在 3.5 版的變更: 可寫入的類位元組物件現在可被接受。

認証處理

ssl.**cert_time_to_seconds**(*cert_time*)

回傳自紀元以來的秒數，給定的 `cert_time` 字串表示憑證的”notBefore” 或”notAfter” 日期，字串🄵用 `"%b %d %H:%M:%S %Y %Z"` 格式（C 語言區域設定）。

以下是一個範例:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

”notBefore” 或”notAfter” 日期必須使用 GMT (**RFC 5280**)。

在 3.5 版的變更: 將輸入的時間直譯🄵 UTC 時間，如輸入字串中指定的’GMT’ 時區。在之前是使用本地的時區。回傳一個整數（在輸入格式中不包括秒的小數部分）。

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

輸入使用 SSL 保護的伺服器的地址 `addr`，輸入形式🄵一個 pair (*hostname*, *port-number*)，取得該伺服器的憑證，🄵以 PEM 編碼字串的形式回傳。如果指定了 `ssl_version`，則使用指定的 SSL 協定來嘗試與伺服器連🄵。如果指定 *ca_certs*，則它應該是一個包含根憑證列表的檔案，🄵與*SSLContext.load_verify_locations()* 中的參數 *cafile* 所使用的格式相同。此呼叫將嘗試使用該組根憑證對伺服器憑證進行驗證，如果驗證失敗，呼叫將失敗。可以使用 `timeout` 參數指定超時時間。

在 3.3 版的變更: 此函式現在是與 IPv6 相容的。

在 3.5 版的變更: 預設的 *ssl_version* 已經從*PROTOCOL_SSLv3* 改🄵*PROTOCOL_TLS*，已確保與現今的伺服器有最大的相容性。

在 3.10 版的變更: 新增 *timeout* 參數。

ssl.**DER_cert_to_PEM_cert**(*der_cert_bytes*)

給定一個以 DER 編碼的位元組 blob 作🄵憑證，回傳以 PEM 編碼字串版本的相同憑證。

ssl.**PEM_cert_to_DER_cert**(*pem_cert_string*)

給定一個以 ASCII PEM 的字串作🄵憑證，回傳以 DER 編碼的位元組序列的相同憑證。

ssl.**get_default_verify_paths**()

回傳一個具有 OpenSSL 的預設 cafile 和 capath 路徑的附名元組。這些路徑與*SSLContext.set_default_verify_paths()* 使用的相同。回傳值是一個*named tuple* `DefaultVerifyPaths`:

- `cafile` - 解析後的 cafile 路徑，如果檔案不存在則🄵 `None`,
- `capath` - 解析後的 capath 路徑，如果目🄵不存在則🄵 `None`,
- `openssl_cafile_env` - 指向 cafile 的 OpenSSL 環境密鑰,
- `openssl_cafile` - hard coded 的 cafile 路徑,
- `openssl_capath_env` - 指向 capath 的 OpenSSL 環境密鑰,
- `openssl_capath` - hard coded 的 capath 目🄵路徑

在 3.4 版被加入.

ssl.**enum_certificates**(*store_name*)

從 Windows 的系統憑證儲存庫中搜尋憑證。*store_name* 可以是 `CA`、`ROOT` 或 `MY` 的其中一個。Windows 也可能會提供額外的憑證儲存庫。

此函式會回傳一個元組 (cert_bytes, encoding_type, trust) 串列。encoding_type 指定了 cert_bytes 的編碼格式。它可以是用來表示 X.509 ASN.1 資料的 `x509_asn` 或是用來表示 PKCS#7 ASN.1 資料的 `pkcs_7_asn`。Trust 通過一組 OIDS 來指定憑證的用途，或是如果憑證對所有用途都可以使用則回傳 `True`。

範例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

可用性: Windows.

在 3.4 版被加入.

ssl.**enum_crls**(*store_name*)

從 Windows 的系統憑證儲存庫中搜尋 CRLs。*store_name* 可以是 `CA`、`ROOT` 或 `MY` 的其中一個。Windows 也可能會提供額外的憑證儲存庫。

此函式會回傳一個元組 (cert_bytes, encoding_type, trust) 串列。encoding_type 指定了 cert_bytes 的編碼格式。它可以是用來表示 X.509 ASN.1 資料的 `x509_asn` 或是用來表示 PKCS#7 ASN.1 資料的 `pkcs_7_asn`。

可用性: Windows.

在 3.4 版被加入.

常數

所有的常數現在都是*enum.IntEnum* 或*enum.IntFlag* 的集合。

在 3.6 版被加入.

ssl.**CERT_NONE**

SSLContext.verify_mode 可能的值。除了*SSLContext.verify_mode* 外，這是預設的模式。對於用🄵端的 sockets，幾乎任何憑證都能被允許。驗證錯誤，像是不被信任或是過期的憑證，會被忽略🄵不會中止 TLS/SSL 握手。

在伺服器模式下，不會從用🄵端請求任何憑證，所以用🄵端不用發送任何用於用🄵端憑證身分驗證的憑證。

參🄵下方*Security considerations* 的討論。

ssl.**CERT_OPTIONAL**

SSLContext.verify_mode 可能的值。在用🄵端模式下，*CERT_OPTIONAL* 具有與*CERT_REQUIRED* 相同的含意。對於客🄵端 sockets 推薦改用*CERT_REQUIRED*。

在伺服器模式下，客🄵憑證請求會被發送給用🄵端。用🄵端可以選擇忽略請求或是選擇發送憑證來執行 TLS 用🄵端憑證身分驗證。如果用🄵端選擇發送憑證，則會對其進行驗證。任何驗證錯誤都會立刻終止 TLS 握手。

使用此設定需要將一組有效的 CA 憑證傳送給*SSLContext.load_verify_locations()*。

ssl.**CERT_REQUIRED**

SSLContext.verify_mode 可能的值。在這個模式下，需要從 socket 連🄵的另一端取得憑證；如果未提供憑證或是驗證失敗，則將會導致*SSLError*。此模式**不能**在用🄵端模式下對憑證進行驗證，因🄵它無法去配對主機名稱。*check_hostname* 也必須被開起來來驗證憑證的真實性。*PROTOCOL_TLS_CLIENT* 會使用*CERT_REQUIRED* 🄵預設開🄵*check_hostname*。

對於 socket 伺服器，此模式會提供🄵制的 TLS 用🄵端憑證驗證。用🄵端憑證請求會被發送給用🄵端🄵且用🄵端必須提供有效且被信任的憑證。

使用此設定需要將一組有效的 CA 憑證傳送給*SSLContext.load_verify_locations()*。

class ssl.**VerifyMode**

enum.IntEnum 🄵 CERT_* 常數的一個集合。

在 3.6 版被加入.

ssl.**VERIFY_DEFAULT**

SSLContext.verify_flags 可能的值。在此模式下，不會檢查憑證吊銷列表 (CRLs)。預設的 OpenSSL 🄵不會請求及驗證 CRLs。

在 3.4 版被加入.

ssl.**VERIFY_CRL_CHECK_LEAF**

SSLContext.verify_flags 可能的值。在此模式下，只會檢查同等的憑證而不會去檢查中間的 CA 憑證。此模式需要提供由對等憑證發行者 (它的直接上級 CA) 的有效的 CRL 簽名。如果🄵有用*SSLContext.load_verify_locations* 載入適當的 CRL，則會驗證失敗。

在 3.4 版被加入.

ssl.**VERIFY_CRL_CHECK_CHAIN**

SSLContext.verify_flags 可能的值。在此模式下，會檢查對等憑證🄵中所有憑證的 CRLs。

在 3.4 版被加入.

ssl.**VERIFY_X509_STRICT**

SSLContext.verify_flags 可能的值，用來禁用已損壞的 X.509 憑證的解🄵方法。

在 3.4 版被加入.

ssl.**VERIFY_ALLOW_PROXY_CERTS**

SSLContext.verify_flags 可能的值，用來🄵用憑證代理驗證。

在 3.10 版被加入.

ssl.**VERIFY_X509_TRUSTED_FIRST**

SSLContext.verify_flags 可能的值。它指示 OpenSSL 在構建信任🄵來驗證憑證時會優先使用被信任的憑證。此旗標預設開🄵。

在 3.4.4 版被加入.

ssl.**VERIFY_X509_PARTIAL_CHAIN**

SSLContext.verify_flags 可能的值。它指示 OpenSSL 接受信任儲存中的中間 CAs 作🄵信任錨，就像自簽名的根 CA 憑證。這樣就能去信任中間 CA 所頒發的憑證，而不一定非要去信任其祖先的根 CA。

在 3.10 版被加入.

class ssl.**VerifyFlags**

enum.IntFlag 🄵 VERIFY_* 常數的其中一個集合。

在 3.6 版被加入.

ssl.**PROTOCOL_TLS**

選擇用🄵端及伺服器均可以支援最高協定版本。🄵管名稱只有「TLS」，但實際上「SSL」和「TLS」均可以選擇。

在 3.6 版被加入.

在 3.10 版之後被🄵用: TLS 的用🄵端及伺服器端需要不同的預設值來實現安全通訊。通用的 TLS 協定常數已被廢除，🄵改用*PROTOCOL_TLS_CLIENT* 和*PROTOCOL_TLS_SERVER*。

ssl.**PROTOCOL_TLS_CLIENT**

自動協商用🄵端和伺服器服務器都支援的最高協定版本，🄵配置用🄵端語境連🄵。該協定預設🄵用*CERT_REQUIRED* 和*check_hostname*。

在 3.6 版被加入.

ssl.**PROTOCOL_TLS_SERVER**

自動協商用🄵端和伺服器都支援的最高協定版本，🄵配置用🄵端語境連🄵。

在 3.6 版被加入.

ssl.**PROTOCOL_SSLv23**

PROTOCOL_TLS 的🄵名。

在 3.6 版之後被🄵用: 請改用*PROTOCOL_TLS*。

ssl.**PROTOCOL_SSLv3**

選擇第三版的 SSL 做🄵通道加密協定。

如果 OpenSSL 是用 `no-ssl3` 編譯的，則此項協定無法使用。

> **⚠ 警告**
>
> 第三版的 SSL 是不安全的，🄵烈建議不要使用。

在 3.6 版之後被🄵用: OpenSSL 已經終止了所有特定版本的協定。請改用預設的*PROTOCOL_TLS_SERVER* 協定或帶有*SSLContext.minimum_version* 和*SSLContext.maximum_version* 的*PROTOCOL_TLS_CLIENT*。

ssl.**PROTOCOL_TLSv1**

選擇 1.0 版的 TLS 做🄵通道加密協定。

在 3.6 版之後被🄵用: OpenSSL 已經將所有版本特定的協定🄵用。

ssl.**PROTOCOL_TLSv1_1**

選擇 1.1 版的 TLS 做🄵通道加密協定。只有在 1.0.1 版本以上的 OpenSSL 才可以選用。

在 3.4 版被加入.

在 3.6 版之後被🄵用: OpenSSL 已經將所有版本特定的協定🄴用。

ssl.**PROTOCOL_TLSv1_2**

選擇 1.2 版的 TLS 做🄵通道加密協定。只有在 1.0.1 版本以上的 OpenSSL 才可以選用。

在 3.4 版被加入.

在 3.6 版之後被🄵用: OpenSSL 已經將所有版本特定的協定🄴用。

ssl.**OP_ALL**

🄵用對 SSL 實作時所🄵生的各種錯誤的緩解措施。此選項預設被設定。它不一定設定與 OpenSSL 的 SSL_OP_ALL 常數相同的旗標。

在 3.2 版被加入.

ssl.**OP_NO_SSLv2**

防止 SSLv2 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級 (peer) 選用 SSLv2 做🄵協定版本。

在 3.2 版被加入.

在 3.6 版之後被🄵用: SSLv2 已被🄵用

ssl.**OP_NO_SSLv3**

防止 SSLv3 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級選用 SSLv3 做🄵協定版本。

在 3.2 版被加入.

在 3.6 版之後被🄵用: SSLv3 已被🄵用

ssl.**OP_NO_TLSv1**

防止 TLSv1 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級選用 TLSv1 做🄵協定版本。

在 3.2 版被加入.

在 3.7 版之後被🄵用: 該選項自從 OpenSSL 1.1.0 以後已被🄵用，請改用新的*SSLContext.minimum_version* 及*SSLContext.maximum_version* 代替。

ssl.**OP_NO_TLSv1_1**

防止 TLSv1.1 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級選用 TLSv1.1 做🄵協定版本。只有 1.0.1 版後的 OpenSSL 版本才能使用。

在 3.4 版被加入.

在 3.7 版之後被🄵用: 此選項自 OpenSSL 1.1.0 版已被🄵用。

ssl.**OP_NO_TLSv1_2**

防止 TLSv1.2 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級選用 TLSv1.2 做🄵協定版本。只有 1.0.1 版後的 OpenSSL 版本才能使用。

在 3.4 版被加入.

在 3.7 版之後被🄵用: 此選項自 OpenSSL 1.1.0 版已被🄵用。

ssl.**OP_NO_TLSv1_3**

防止 TLSv1.3 連🄵。此選項只可以跟*PROTOCOL_TLS* 一起使用。它會防止同級選用 TLSv1.3 做🄵協定版本。TSL1.3 只適用於 1.1.1 版以後的 OpenSSL。當使用 Python 編譯舊版的 OpenSSL 時，該標志預設🄵 *0*。

在 3.6.3 版被加入.

在 3.7 版之後被🄵用: 此選項自 OpenSSL 1.1.0 以後已被🄵用。它被添加到 2.7.15 和 3.6.3 中，以向後相容 OpenSSL 1.0.2。

ssl.**OP_NO_RENEGOTIATION**

停用所有在 TLSv1.2 及更早版本的重協商 (renegotiation)。不發送 HelloRequest 訊息，🄵忽略通過 ClientHello 的重協商請求。

此選項僅適用於 OpenSSL 1.1.0h 及更新版本。

在 3.7 版被加入.

ssl.**OP_CIPHER_SERVER_PREFERENCE**

使用伺服器的加密方法名稱字串排序優先順序，而不是用🄴端的。此選項🄵不會影響到用🄴端及 SSLv2 伺服器的 sockets。

在 3.3 版被加入.

ssl.**OP_SINGLE_DH_USE**

防止對不同的 SSL 會談重🄵使用相同的 DH 密鑰。這會加🄴向前保密但需要更多的運算資源。此選項只適用於伺服器 sockets。

在 3.3 版被加入.

ssl.**OP_SINGLE_ECDH_USE**

防止對不同的 SSL 會談重🄵使用相同的 ECDH 密鑰。這會加🄴向前保密但需要更多的運算資源。此選項只適用於伺服器 sockets。

在 3.3 版被加入.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**

在 TLS 1.3 握手中發送🄵擬的變更加密方法規範 (CCS) 消息，以使 TLS 1.3 連接看起來更像 TLS 1.2 連🄵。

此選項僅適用於 OpenSSL 1.1.1 及更新版本。

在 3.8 版被加入.

ssl.**OP_NO_COMPRESSION**

在 SSL 通道上禁用壓縮。如果應用程序協定支援自己的壓縮方案，這會很有用。

在 3.3 版被加入.

class ssl.**Options**

enum.IntFlag 🄵 OP_* 常數中的一個集合。

ssl.**OP_NO_TICKET**

防止用🄴端請求會談票據。

在 3.6 版被加入.

ssl.**OP_IGNORE_UNEXPECTED_EOF**

忽略意外關閉的 TLS 連🄵。

此選項僅適用於 OpenSSL 3.0.0 及更新版本。

在 3.10 版被加入.

ssl.**OP_ENABLE_KTLS**

允許使用 TLS 核心。要想受益於該功能，OpenSSL 必須編譯🄵支援該功能，🄵且想使用的加密套件及擴充套件也必須被該功能支援 (該功能所支援的列表可能會因平台及核心而有所差🄴)。

請注意當允許使用 TLS 核心時，一些加密操作將直接由核心執行而不是經由任何由可用的 OpenSSL 所提供的程序，而這可能🄵非你所想使用的，例如：當應用程式要求所有的加密操作由 FIPS 提供執行。

此選項僅適用於 OpenSSL 3.0.0 及更新版本。

在 3.12 版被加入.

ssl.**OP_LEGACY_SERVER_CONNECT**

只允許 OpenSSL 與未修補的伺服器進行遺留 (legacy) 不安全重協商。

在 3.12 版被加入.

ssl.**HAS_ALPN**

OpenSSL 函式庫是否🄴建支援 應用層協定協商 TLS 擴充套件，該擴充套件描述在 **RFC 7301** 中。

在 3.5 版被加入.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**

OpenSSL 函式庫是否🄵建支援不檢查主題通用名稱及*SSLContext.hostname_checks_common_name* 是否可寫。

在 3.7 版被加入.

ssl.**HAS_ECDH**

OpenSSL 函式庫是否🄴建支援基於橢圓曲🄵的 (Elliptic Curve-based) Diffie-Hellman 金鑰交🄴。此回傳值應該要🄵 true 除非發布者明確禁用此功能。

在 3.3 版被加入.

ssl.**HAS_SNI**

OpenSSL 函式庫是否🄴建支援 伺服器名懲提示擴充套件 (在 **RFC 6066** 中定義)。

在 3.2 版被加入.

ssl.**HAS_NPN**

OpenSSL 函式庫是否🄴建支援 下一代協定協商該功能在應用層協定協商 中有描述。當此值🄵 true 時，你可以使用*SSLContext.set_npn_protocols()* 方法來公告你想支援的協定。

在 3.3 版被加入.

ssl.**HAS_SSLv2**

此 OpenSSL 函式庫是否🄴建支援 SSL 2.0 協定。

在 3.7 版被加入.

ssl.**HAS_SSLv3**

此 OpenSSL 函式庫是否🄴建支援 SSL 3.0 協定。

在 3.7 版被加入.

ssl.**HAS_TLSv1**

此 OpenSSL 函式庫是否🄴建支援 TLS 1.0 協定。

在 3.7 版被加入.

ssl.**HAS_TLSv1_1**

此 OpenSSL 函式庫是否🄴建支援 TLS 1.1 協定。

在 3.7 版被加入.

ssl.**HAS_TLSv1_2**

此 OpenSSL 函式庫是否🄴建支援 TLS 1.2 協定。

在 3.7 版被加入.

ssl.**HAS_TLSv1_3**

此 OpenSSL 函式庫是否🄴建支援 TLS 1.3 協定。

在 3.7 版被加入.

ssl.**HAS_PSK**

此 OpenSSL 函式庫是否🄴建支援 TLS-PSK。

在 3.13 版被加入.

ssl.**HAS_PHA**

此 OpenSSL 函式庫是否🄴建支援 TLS-PHA。

在 3.14 版被加入.

ssl.**CHANNEL_BINDING_TYPES**

支援的 TLS 通道綁定類型列表。列表中的字串可以作🄵*SSLSocket.get_channel_binding()* 的參數。

在 3.3 版被加入.

ssl.**OPENSSL_VERSION**

直譯器所加載的 OpenSSL 函式庫的版本字串:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

在 3.2 版被加入.

ssl.**OPENSSL_VERSION_INFO**

代表 OpenSSL 函式庫版本資訊的五個整數的元組:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

在 3.2 版被加入.

ssl.**OPENSSL_VERSION_NUMBER**

OpenSSL 函式庫的初始版本，以單一整數表示:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

在 3.2 版被加入.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**

ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**

ALERT_DESCRIPTION_*

來自 **RFC 5246** 和其他文件的警報描述。IANA TLS Alert Registry 包含了此列表以及其含義定義所在的 RFC 的引用。

被用來做🄵*SSLContext.set_servername_callback()* 中回呼函式的回傳值。

在 3.4 版被加入.

class ssl.**AlertDescription**

enum.IntEnum 🄵 ALERT_DESCRIPTION_* 常數中的一個集合。

在 3.6 版被加入.

Purpose.**SERVER_AUTH**

create_default_context() 和*SSLContext.load_default_certs()* 的選項。此值表示該語境可能會用於驗證網頁伺服器 (因此它將用於建立用🄵端 socket)。

在 3.4 版被加入.

Purpose.**CLIENT_AUTH**

create_default_context() 和*SSLContext.load_default_certs()* 的選項。此值表示該語境可能會用於驗證網頁用🄵端 (因此，它將用於建立伺服器端的 socket)。

在 3.4 版被加入.

class ssl.**SSLErrorNumber**

enum.IntEnum 🄵 SSL_ERROR_* 常數中的一個集合。

在 3.6 版被加入.

class ssl.**TLSVersion**

用於*SSLContext.maximum_version* 和*SSLContext.minimum_version* 的 SSL 和 TLS 版本*enum.IntEnum* 集合。

在 3.7 版被加入.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**

最低或最高支援的 SSL 或 TLS 版本。這些是特殊常數。它們的值🄵不反映可用的最低和最高 TLS/SSL 版本。

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**

SSL 3.0 到 TLS 1.3。

在 3.10 版之後被🄵用: 除了*TLSVersion.TLSv1_2* 和*TLSVersion.TLSv1_3* 之外，所有的*TLSVersion* 成員都已被🄵用。

19.3.2 SSL Sockets

class ssl.**SSLSocket**(*socket.socket*)

SSL sockets 提供以下*Socket 物件* 方法:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*、*getsockname()*
- *getsockopt()*、*setsockopt()*
- *gettimeout()*、*settimeout()*、*setblocking()*
- *listen()*
- *makefile()*
- *recv()*、*recv_into()* （但不允許傳遞非零的 `flags` 引數）
- *send()*、*sendall()* （同樣不允許傳遞非零的 `flags` 引數）
- *sendfile()* （但*os.sendfile* 只能用於純文本 sockets，其余則會使用*send()*）
- *shutdown()*

然而，由於 SSL（和 TLS）協定在 TCP 之上有自己的框架，因此在某些方面，SSL sockets 的抽象可能會與普通作業系統級🄵的 sockets 規範有所不同。特🄵是請參🄵關於 *non-blocking sockets* 的🄵明。

SSLSocket 的實例必須使用*SSLContext.wrap_socket()* 方法建立。

在 3.5 版的變更: 新增 `sendfile()` 方法。

在 3.5 版的變更: `shutdown()` 不會在每次接收或發送位元組時重置 socket 超時時間。現在，socket 超時時間是關閉操作的最大總持續時間。

在 3.6 版之後被🄵用: 直接建立*SSLSocket* 實例的方式已被🄵用，請使用*SSLContext.wrap_socket()* 來包裝 socket。

在 3.7 版的變更: *SSLSocket* 實例必須使用*wrap_socket()* 建立。在較早的版本中可以直接建立實例，但這從未被記🄵或正式支援。

在 3.10 版的變更: Python 現在🄵部使用了 `SSL_read_ex` 和 `SSL_write_ex` 函式。這些函式支援讀取和寫入大於 2 GB 的資料。寫入零長度的資料不再會導致協定違規錯誤。

SSL sockets 還具有以下附加方法和屬性:

`SSLSocket.`**`read`**`(`*len=1024*, *buffer=None*`)`

從 SSL socket 讀取 *len* 位元組的資料，🄵將結果以 `bytes` 實例的形式回傳。如果指定了 *buffer*，則將資料讀入緩衝區，🄵回傳讀取的位元組。

如果 socket 是非阻塞的則會引發*SSLWantReadError* 或*SSLWantWriteError* 🄵且讀取操作將會被阻塞。

由於在任何時刻都可能發生重新協商，呼叫*read()* 也可能觸發寫入操作。

在 3.5 版的變更: 當接收或發送位元組時，socket 的超時時間將不再重置。現在，socket 超時時間是讀取最多 *len* 位元組的總最大持續時間。

在 3.6 版之後被🄵用: 請改用 `recv()` 來替🄵掉*read()*。

`SSLSocket.`**`write`**`(`*data*`)`

Write *data* to the SSL socket and return the number of bytes written. The *data* argument must be an object supporting the buffer interface.

如果 socket 是非阻塞的則會引發*SSLWantReadError* 或*SSLWantWriteError* 🄵且寫入操作將會被阻塞。

由於在任何時刻都可能發生重新協商，呼叫*write()* 也可能觸發讀取操作。

在 3.5 版的變更: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *data*.

在 3.6 版之後被🄵用: 請改用 `send()` 來替🄵掉*write()*。

備🄵

read() 和*write()* 方法🄵低階層的方法，負責讀取和寫入未加密的應用層資料，🄵將其加密/解密🄵加密的寫入層資料。這些方法需要一個已建立的 SSL 連接，即握手已完成，且未呼叫*SSLSocket.unwrap()*。

通常你應該使用像*recv()* 和*send()* 這樣的 socket API 方法，而不是直接使用這些方法。

`SSLSocket.`**`do_handshake`**`(`*block=False*`)`

執行 SSL 設定握手。

If *block* is true and the timeout obtained by *gettimeout()* is zero, the socket is set in blocking mode until the handshake is performed.

在 3.4 版的變更: The handshake method also performs `match_hostname()` when the *check_hostname* attribute of the socket's *context* is true.

在 3.5 版的變更: Socket 超時時間已經不會在每次接收或傳送位元組時重置。現在，超時時間是握手過程的最大總持續時間。

在 3.7 版的變更: Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

`SSLSocket.`**`getpeercert`**(*binary_form=False*)

如果連🄵端🄵有證書，則回傳 `None`。如果 SSL 握手尚未完成，則引發*`ValueError`*。

如果 `binary_form` 參數🄵*`False`*，且從對等 (peer) 接收到證書，則該方法回傳一個*`dict`* 實例。如果證書未被驗證，則該字典🄵空。若證書已被驗證，則回傳的字典將包含數個鍵值，包括 `subject` (證書所簽發的對象) 和 `issuer` (簽發證書的主體)。如果證書中包含 *Subject Alternative Name* 擴充 (參考**RFC 3280**)，字典中還會有一個 `subjectAltName` 鍵。

`subject` 和 `issuer` 欄位欄位是包含相對識🄵名稱 (relative distinguished names, RDNs) 序列的元組，這些 RDN 來自證書資料結構中的相應欄位。每個 RDN 都是一組名稱與值的對。以下是現實中的範例：

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

如果 `binary_form` 參數設定🄵*`True`*，且對等提供了證書，則該方法會以 DER 編碼形式將整個證書以位元組序列形式回傳。如果對等未提供證書，則回傳*`None`*。對等是否提供證書取🄵於 SSL socket 的🄵色：

- 對於用🄵端 SSL socket，伺服器將永遠提供證書，無論是否需要進行驗證；
- 對於伺服器 SSL socket，用🄵端僅在伺服器要求時才會提供證書；因此，如果你使用的是*`CERT_NONE`* (而非*`CERT_OPTIONAL`* 或*`CERT_REQUIRED`*)，則*`getpeercert()`* 會回傳*`None`*。

請見*`SSLContext.check_hostname`*。

在 3.2 版的變更: The returned dictionary includes additional items such as `issuer` and `notBefore`.

在 3.4 版的變更: *`ValueError`* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

在 3.9 版的變更: IPv6 address strings no longer have a trailing new line.

`SSLSocket.`**`get_verified_chain`**()

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as *`get_unverified_chain()`*.

在 3.13 版被加入.

`SSLSocket.`**`get_unverified_chain`**()

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

在 3.13 版被加入.

SSLSocket.**cipher**()

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

SSLSocket.**shared_ciphers**()

Return the list of ciphers available in both the client and server. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns `None` if no connection has been established or the socket is a client socket.

在 3.5 版被加入.

SSLSocket.**compression**()

Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

在 3.3 版被加入.

SSLSocket.**get_channel_binding**(*cb_type='tls-unique'*)

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

在 3.3 版被加入.

SSLSocket.**selected_alpn_protocol**()

Return the protocol that was selected during the TLS handshake. If *SSLContext.set_alpn_protocols()* was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

在 3.5 版被加入.

SSLSocket.**selected_npn_protocol**()

Return the higher-level protocol that was selected during the TLS/SSL handshake. If *SSLContext.set_npn_protocols()* was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

在 3.3 版被加入.

在 3.10 版之後被🄵用: NPN has been superseded by ALPN

SSLSocket.**unwrap**()

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket.**verify_client_post_handshake**()

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see *SSLContext.post_handshake_auth*.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an *SSLError* is raised.

> **備🄵**
>
> Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises *NotImplementedError*.

在 3.8 版被加入.

`SSLSocket.`**`version`**`()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

在 3.5 版被加入.

`SSLSocket.`**`pending`**`()`

Returns the number of already decrypted bytes available for read, pending on the connection.

`SSLSocket.`**`context`**

The *SSLContext* object this SSL socket is tied to.

在 3.2 版被加入.

`SSLSocket.`**`server_side`**

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

在 3.2 版被加入.

`SSLSocket.`**`server_hostname`**

Hostname of the server: *str* type, or `None` for server-side socket or if the hostname was not specified in the constructor.

在 3.2 版被加入.

在 3.7 版的變更: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

`SSLSocket.`**`session`**

The *SSLSession* for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before *do_handshake()* has been called to reuse a session.

在 3.6 版被加入.

`SSLSocket.`**`session_reused`**

在 3.6 版被加入.

19.3.3 SSL Contexts

在 3.2 版被加入.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

`class` `ssl.`**`SSLContext`**(*protocol=None*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	yes	no	no[1]	no	no	no
SSLv3	no	yes	no[2]	no	no	no
TLS (SSLv23)[3]	no[1]	no[2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

也參考

`create_default_context()` lets the `ssl` module choose security settings for a given purpose.

在 3.6 版的變更: The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only `HIGH` ciphers, no `NULL` ciphers and no `MD5` ciphers.

在 3.10 版之後被🄴用: `SSLContext` without protocol argument is deprecated. The context class will either require `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol in the future.

在 3.10 版的變更: The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER` use TLS 1.2 as minimum TLS version.

備🄴

`SSLContext` only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

備🄴

`SSLContext` is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

`SSLContext` 物件具有以下方法和屬性:

`SSLContext.`**`cert_store_stats`**`()`

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

在 3.4 版被加入.

[3] TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

[1] `SSLContext` 預設會關閉 SSLv2 的`OP_NO_SSLv2`。

[2] `SSLContext` 預設會關閉 SSLv3 的`OP_NO_SSLv3`。

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn't match with the certificate.

在 3.3 版的變更: 新增可選引數 *password*。

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings *Purpose.SERVER_AUTH* loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

在 3.4 版被加入.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

在 3.4 版的變更: 新增可選引數 *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

> **備🄵**
>
> Certificates in a capath directory aren't loaded unless they have been used at least once.

在 3.4 版被加入.

SSLContext.**get_ciphers**()

Get a list of enabled ciphers. The list is in order of cipher priority. See *SSLContext.set_ciphers()*.

範例:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

在 3.6 版被加入.

SSLContext.**set_default_verify_paths**()

Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.**set_ciphers**(*ciphers*, /)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

> **備🄵**
>
> when connected, the *SSLSocket.cipher()* method of SSL sockets will give the currently selected cipher.
>
> TLS 1.3 cipher suites cannot be disabled with *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*alpn_protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the *SSLSocket.selected_alpn_protocol()* method will return the agreed-upon protocol.

此方法會在*HAS_ALPN* 🄵 `False` 時引發*NotImplementedError*。

在 3.5 版被加入.

SSLContext.**set_npn_protocols**(*npn_protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

此方法會在*HAS_NPN* 🄵 `False` 時引發*NotImplementedError*。

在 3.3 版被加入.

在 3.10 版之後被🄵用: NPN has been superseded by ALPN

SSLContext.**sni_callback**

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the *ssl.SSLSocket*'s *SSLSocket.context* attribute to a new object of type *SSLContext* representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. The *SSLSocket.getpeercert()*, *SSLSocket.get_verified_chain()*, *SSLSocket.get_unverified_chain()* *SSLSocket.cipher()* and *SSLSocket.compression()* methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

This method will raise *NotImplementedError* if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

在 3.7 版被加入.

SSLContext.**set_servername_callback**(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use *sni_callback* instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is a decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

在 3.4 版被加入.

SSLContext.**load_dh_params**(*dhfile*, */*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_DH_USE* option to further improve security.

在 3.3 版被加入.

SSLContext.**set_ecdh_curve**(*curve_name*, /)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_ECDH_USE* option to further improve security.

此方法在*HAS_ECDH* 🄵 `False` 時不可用。

在 3.3 版被加入.

也參考

SSL/TLS & Perfect Forward Secrecy
Vincent Bernat.

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class* (default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise *SSLError*.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the *SSLSocket.do_handshake()* method. Calling *SSLSocket.do_handshake()* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *False*, it will raise the exceptions back to the caller.

session, see *session*.

To wrap an *SSLSocket* in another *SSLSocket*, use *SSLContext.wrap_bio()*.

在 3.5 版的變更: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

在 3.6 版的變更: 新增 *session* 引數。

在 3.7 版的變更: The method returns an instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

SSLContext.**sslsocket_class**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be assigned to on instances of *SSLContext* in order to return a custom subclass of *SSLSocket*.

在 3.7 版被加入.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

在 3.6 版的變更: 新增 *session* 引數。

在 3.7 版的變更: The method returns an instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

SSLContext.**sslobject_class**

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

在 3.7 版被加入.

SSLContext.**session_stats**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

Whether to match the peer cert's hostname in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled. The *PROTOCOL_TLS_CLIENT* protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

範例:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

在 3.4 版被加入.

在 3.7 版的變更: *verify_mode* is now automatically changed to *CERT_REQUIRED* when hostname checking is enabled and *verify_mode* is *CERT_NONE*. Previously the same operation would have failed with a *ValueError*.

SSLContext.**keylog_filename**

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as

Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

在 3.8 版被加入.

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to *TLSVersion.MAXIMUM_SUPPORTED*. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

The attributes *maximum_version*, *minimum_version* and *SSLContext.options* all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with *OP_NO_TLSv1_2* in *options* and *maximum_version* set to *TLSVersion.TLSv1_2* will not be able to establish a TLS 1.2 connection.

在 3.7 版被加入.

SSLContext.**minimum_version**

Like *SSLContext.maximum_version* except it is the lowest supported version or *TLSVersion.MINIMUM_SUPPORTED*.

在 3.7 版被加入.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a *PROTOCOL_TLS_SERVER* context. The setting has no impact on TLS 1.0 to 1.2 connections.

在 3.8 版被加入.

SSLContext.**options**

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

在 3.6 版的變更: *SSLContext.options* 會回傳*Options* 旗標:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

在 3.7 版之後被🄵用: All `OP_NO_SSL*` and `OP_NO_TLS*` options have been deprecated since Python 3.7. Use *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

SSLContext.**post_handshake_auth**

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

在 3.8 版被加入.

SSLContext.**protocol**

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.**hostname_checks_common_name**

Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

在 3.7 版被加入.

在 3.10 版的變更: The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.**security_level**

An integer representing the security level for the context. This attribute is read-only.

在 3.10 版被加入.

SSLContext.**verify_flags**

The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

在 3.4 版被加入.

在 3.6 版的變更: *SSLContext.verify_flags* 會回傳*VerifyFlags* 旗標:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

在 3.6 版的變更: *SSLContext.verify_mode* 會回傳*VerifyMode* 列舉:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.**set_psk_client_callback**(*callback*)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to `256` octets when UTF-8 encoded. PSK is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

備🄵

使用 TLS 1.3 時:

- `hint` 參數始終🄵*None*。
- client-identity must be a non-empty string.

範例用法:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
```

(繼續下一頁)

(繼續上一頁)

```
}
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

在 3.13 版被加入.

`SSLContext.`**`set_psk_server_callback`**(*callback*, *identity_hint=None*)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to `256` octets when UTF-8 encoded.

> **備🄵**
>
> When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

範例用法:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

在 3.13 版被加入.

19.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer

knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling *`SSLContext.load_default_certs()`*, this is done automatically with *`create_default_context()`*.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to *`SSLContext.load_cert_chain()`* needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
..............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

19.3.5 范例

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification.

verify_mode is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
```

(繼續下一頁)

(繼續上一頁)

```
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

參🄵下方*Security considerations* 的討論。

伺服器端操作

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *`SSLContext.wrap_socket()`* method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

19.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most *SSLSocket* methods will raise either *SSLWantWriteError* or *SSLWantReadError* instead of *BlockingIOError* if an I/O operation would block. *SSLWantReadError* will be raised if a read operation on the underlying socket is necessary, and *SSLWantWriteError* for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

  在 3.5 版的變更: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising *SSLWantWriteError* or *SSLWantReadError*.

- Calling *select()* tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to *select()*.

- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without *select()* being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a *select()* call if still necessary.

  (of course, similar provisions apply when using other primitives such as *poll()*, or those in the *selectors* module)

- The SSL handshake itself will be non-blocking: the *SSLSocket.do_handshake()* method has to be retried until it returns successfully. Here is a synopsis using *select()* to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

也參考

The *asyncio* module supports *non-blocking SSL sockets* and provides a higher level *Streams API*. It polls for events using the *selectors* module and handles *SSLWantWriteError*, *SSLWantReadError* and *BlockingIOError* exceptions. It runs the SSL handshake asynchronously as well.

19.3.7 記憶體 BIO 支援

在 3.5 版被加入.

Ever since the SSL module was introduced in Python 2.6, the *SSLSocket* class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by *socket.socket*, from which *SSLSocket* also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the "select/poll on a file descriptor" (readiness based) model that is assumed by *socket.socket* and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of *SSLSocket* called *SSLObject* is provided.

class ssl.**SSLObject**

A reduced-scope variant of *SSLSocket* representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate "BIO" objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An *SSLObject* instance must be created using the *wrap_bio()* method. This method will create the *SSLObject* instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *get_verified_chain()*
- *get_unverified_chain()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

When compared to *SSLSocket*, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying *MemoryBIO* buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call *do_handshake()* to start the handshake.

- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.
- The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject`:

- All IO on an `SSLObject` is *non-blocking*. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.

在 3.7 版的變更: `SSLObject` instances must be created with `wrap_bio()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class ssl.**MemoryBIO**

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*, /)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*, /)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call `write()`. The attribute `eof` will become true after all data currently in the buffer has been read.

19.3.8 SSL session

在 3.6 版被加入.

class ssl.**SSLSession**

Session object used by `session`.

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the `create_default_context()` function to create your SSL context. It will load the system's trusted CA

certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the *smtplib.SMTP* class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

手動設定

驗証憑証

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of the time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

在 3.7 版的變更: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

協定版本

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()* or *RAND_bytes()* is sufficient.

19.3.10 TLS 1.3

在 3.7 版被加入.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

也參考

socket.socket 類🄵
底層*socket* 類🄵的文件

SSL/TLS Strong Encryption: An Introduction
Apache HTTP Server 文件的介紹

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 4086: Randomness Requirements for Security
Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2
T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters
IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

Mozilla's Server Side TLS recommendations
Mozilla

19.4 `select` --- 等待 I/O 完成

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

備🄵

The *selectors* module allows high-level and efficient I/O multiplexing, built upon the *select* module primitives. Users are encouraged to use the *selectors* module instead, unless they want precise control over the OS-level primitives used.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

The module defines the following:

exception `select.error`

A deprecated alias of *OSError*.

在 3.3 版的變更: Following **PEP 3151**, this class was made an alias of *OSError*.

`select.`**`devpoll`**`()`

(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

在 3.3 版被加入.

在 3.4 版的變更: The new file descriptor is now non-inheritable.

`select.`**`epoll`**`(`*sizehint=-1*, *flags=0*`)`

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

在 3.3 版的變更: 新增 *flags* 參數。

在 3.4 版的變更: Support for the `with` statement was added. The new file descriptor is now non-inheritable.

在 3.4 版之後被🄵用: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use *os.set_inheritable()* to make the file descriptor inheritable.

`select.poll()`

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section *Polling Objects* below for the methods supported by polling objects.

`select.kqueue()`

(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

在 3.4 版的變更: The new file descriptor is now non-inheritable.

`select.kevent`(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

`select.select`(*rlist*, *wlist*, *xlist*[, *timeout*])

This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating-point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python *file objects* (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

> **備🄵**
>
> File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

在 3.5 版的變更: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

`select.PIPE_BUF`

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by `select()`, `poll()` or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

可用性: Unix

在 3.2 版被加入.

19.4.1 `/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is *O*(*highest file descriptor*) and `poll()` is *O*(*number of file descriptors*), `/dev/poll` is *O*(*active file descriptors*).

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.close()`

Close the file descriptor of the polling object.

在 3.4 版被加入.

`devpoll.closed`

`True` if the polling object is closed.

在 3.4 版被加入.

`devpoll.fileno()`

Return the file descriptor number of the polling object.

在 3.4 版被加入.

`devpoll.register`(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

> **警告**
>
> Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

`devpoll.modify`(*fd*[, *eventmask*])

This method does an *unregister()* followed by a *register()*. It is (a bit) more efficient that doing the same explicitly.

`devpoll.unregister`(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

`devpoll.poll`([*timeout*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor --- `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *None*, the call will block until there is an event for this poll object.

在 3.5 版的變更: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

常數	含義
EPOLLIN	Available for read
EPOLLOUT	Available for write
EPOLLPRI	Urgent data for read
EPOLLERR	Error condition happened on the assoc. fd
EPOLLHUP	Hang up happened on the assoc. fd
EPOLLET	Set Edge Trigger behavior, the default is Level Trigger behavior
EPOLLONESH	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
EPOLLEXCLU	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
EPOLLRDHUP	Stream socket peer closed connection or shut down writing half of connection.
EPOLLRDNORM	等價於 EPOLLIN
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORM	等價於 EPOLLOUT
EPOLLWRBAND	Priority data may be written.
EPOLLMSG	Ignored.
EPOLLWAKEUP	Prevents sleep during event waiting.

在 3.6 版被加入: EPOLLEXCLUSIVE was added. It's only supported by Linux Kernel 4.5 or later.

在 3.14 版被加入: EPOLLWAKEUP was added. It's only supported by Linux Kernel 3.5 or later.

epoll.**close**()

Close the control file descriptor of the epoll object.

epoll.**closed**

True if the epoll object is closed.

epoll.**fileno**()

Return the file descriptor number of the control fd.

epoll.**fromfd**(*fd*)

Create an epoll object from a given file descriptor.

epoll.**register**(*fd*[, *eventmask*])

Register a fd descriptor with the epoll object.

epoll.**modify**(*fd*, *eventmask*)

Modify a registered file descriptor.

epoll.**unregister**(*fd*)

Remove a registered file descriptor from the epoll object.

在 3.9 版的變更: The method no longer ignores the *EBADF* error.

epoll.**poll**(*timeout=None*, *maxevents=-1*)

Wait for events. timeout in seconds (float)

在 3.5 版的變更: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is *O*(*highest file descriptor*), while `poll()` is *O*(*number of file descriptors*).

`poll.`**`register`**`(`*fd*[, *eventmask*]`)`

Register a file descriptor with the polling object. Future calls to the *`poll()`* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *`fileno()`* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

常數	含義
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output: writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.`**`modify`**`(`*fd*, *eventmask*`)`

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *`OSError`* exception with errno `ENOENT` to be raised.

`poll.`**`unregister`**`(`*fd*`)`

Remove a file descriptor being tracked by a polling object. Just like the *`register()`* method, *fd* can be an integer or an object with a *`fileno()`* method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a *`KeyError`* exception to be raised.

`poll.`**`poll`**`(`[*timeout*]`)`

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor --- `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *`None`*, the call will block until there is an event for this poll object.

在 3.5 版的變更: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *`InterruptedError`*.

19.4.4 Kqueue Objects

`kqueue.`**`close`**`()`

Close the control file descriptor of the kqueue object.

`kqueue.`**`closed`**

`True` if the kqueue object is closed.

kqueue.**fileno**()

Return the file descriptor number of the control fd.

kqueue.**fromfd**(*fd*)

Create a kqueue object from a given file descriptor.

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist

Low level interface to kevent

- changelist must be an iterable of kevent objects or None
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible); the default is None, to wait forever

在 3.5 版的變更: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.5 Kevent Objects

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *fileno()* method. kevent stores the integer internally.

kevent.**filter**

Name of the kernel filter.

常數	含義
KQ_FILTER_READ	Takes a descriptor and returns whenever there is data available to read
KQ_FILTER_WRITE	Takes a descriptor and returns whenever there is data available to write
KQ_FILTER_AIO	AIO requests
KQ_FILTER_VNODE	Returns when one or more of the requested events watched in *fflag* occurs
KQ_FILTER_PROC	Watch for events on a process id
KQ_FILTER_NETDEV	Watch for events on a network device [not available on macOS]
KQ_FILTER_SIGNAL	Returns whenever the watched signal is delivered to the process
KQ_FILTER_TIMER	Establishes an arbitrary timer

kevent.**flags**

Filter action.

常數	含義
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

kevent.**fflags**

Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

常數	含義
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags:

常數	含義
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

常數	含義
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on macOS):

常數	含義
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.**data**

Filter specific data.

kevent.**udata**

User defined value.

19.5 selectors --- 高階 I/O 多工

在 3.4 版被加入.

原始碼： Lib/selectors.py

19.5.1 簡介

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, "file object" refers to any object with a `fileno()` method, or a raw file descriptor. See *file object*.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

備🄵

The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

也參考

select
Low-level I/O multiplexing module.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

19.5.2 Classes

Classes hierarchy:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

常數	含義
`selectors.`**`EVENT_READ`**	Available for read
`selectors.`**`EVENT_WRITE`**	Available for write

class `selectors.`**SelectorKey**

A `SelectorKey` is a `namedtuple` used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several `BaseSelector` methods.

fileobj
File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.**BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector*, *KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the *context manager* protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated *SelectorKey* instance, or raises a *KeyError* if *fileobj* is not registered. It will raise *ValueError* if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)` followed by `BaseSelector.register(fileobj, events, data)`, except that it can be implemented more efficiently.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is not registered.

abstractmethod select(*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the *SelectorKey* instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

> **備🄵**
>
> This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

在 3.5 版的變更: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** for the rationale), instead of returning an empty list of events before the timeout.

close()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)

Return the key associated with a registered file object.

This returns the *SelectorKey* instance associated to this file object, or raises *KeyError* if the file object is not registered.

abstractmethod get_map()

Return a mapping of file objects to selector keys.

This returns a *Mapping* instance mapping registered file objects to their associated *SelectorKey* instance.

class selectors.**DefaultSelector**

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.**SelectSelector**

select.select()-based selector.

class selectors.**PollSelector**

select.poll()-based selector.

class selectors.**EpollSelector**

select.epoll()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.epoll()* object.

class selectors.**DevpollSelector**

select.devpoll()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.devpoll()* object.

在 3.5 版被加入.

class selectors.**KqueueSelector**

select.kqueue()-based selector.

fileno()

This returns the file descriptor used by the underlying *select.kqueue()* object.

19.5.3 范例

Here is a simple echo server implementation:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
```

(繼續下一頁)

(繼續上一頁)

```
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 `signal` --- 設定非同步事件的處理函式

原始碼： Lib/signal.py

本模組提供於 Python 中使用訊號處理程式的機制。

19.6.1 一般規則

signal.signal() 函式允許定義自訂的處理程式，會在收到訊號時執行。我們安裝了少數的預設處理程式：*SIGPIPE* 會被忽略 (所以管道和 socket 上的寫入錯誤可以當作一般的 Python 例外報告)，而*SIGINT* (如果父行程🄵有改變它的話) 會被轉🄵成*KeyboardInterrupt* 例外。

特定訊號的處理程式一旦被設定，就會一直被安裝，直到被明確地重設🄵止 (不管底層的實作🄵何，Python 皆模擬出 BSD 風格的介面)，但*SIGCHLD* 的處理程式除外，它會跟隨底層的實作。

在 WebAssembly 平台上，訊號是模擬出來的，故行🄵不同。有幾個函式和訊號在這些平台上是不可用的。

Python 訊號處理程式的執行

Python 訊號處理程式不會在低階（C 語言）訊號處理程式中執行。相反地，低階訊號處理程式會設定一個旗標，告訴🄵擬機在稍後執行相對應的 Python 訊號處理程式（例如在下一個*bytecode* 指令）。這會有一些後果：

- 捕捉像*SIGFPE* 或*SIGSEGV* 這類由 C 程式碼中無效操作所引起的同步錯誤是🄵有意義的。Python 將從訊號處理程式中回傳到 C 程式碼，而 C 程式碼很可能再次引發相同的訊號，導致 Python 明顯假當機 (hang)。從 Python 3.3 開始，你可以使用*faulthandler* 模組來報告同步錯誤。
- 純粹以 C 實作的長時間計算（例如在大量文字上的正規表示式比對）可能會不間斷地運行任意長度的時間而不考慮收到的任何訊號。當計算完成時，Python 訊號處理程式會被呼叫。
- 如果處理程式引發例外，就會在主執行緒中「憑空」🄵生例外。請參🄵下面的🄵明。

訊號和執行緒

Python 訊號處理程式總是在主直譯器的主 Python 執行緒中執行，即使訊號是在另一個執行緒中接收到的。這意味著訊號不能用來做🄴執行緒間通訊的方式。你可以使用*threading* 模組的同步原語 (synchronization primitive) 來代替。

此外，只有主直譯器的主執行緒才被允許設定新的訊號處理程式。

19.6.2 模組🄵容

在 3.5 版的變更: 下面列出的訊號 (SIG*)、處理器 (*SIG_DFL*、*SIG_IGN*) 和訊號遮罩 (sigmask) (*SIG_BLOCK*、*SIG_UNBLOCK*、*SIG_SETMASK*) 的相關常數被轉🄵成*enums* (*Signals*、*Handlers* 和*Sigmasks*) 。*getsignal()*、*pthread_sigmask()*、*sigpending()* 和*sigwait()* 函式會回傳可被人類🄵讀的枚舉作🄵*Signals* 物件。

訊號模組定義了三個枚舉:

class signal.**Signals**

SIG* 常數和 CTRL_* 常數的*enum.IntEnum* 集合。

在 3.5 版被加入.

class signal.**Handlers**

SIG_DFL 和*SIG_IGN* 常數的*enum.IntEnum* 集合。

在 3.5 版被加入.

class signal.**Sigmasks**

SIG_BLOCK、*SIG_UNBLOCK* 和*SIG_SETMASK* 常數的*enum.IntEnum* 集合。

可用性: Unix.

更多資訊請見 *sigprocmask(2)* 與 *pthread_sigmask(3)* 🄵上手🄵。

在 3.5 版被加入.

在*signal* 模組中定義的變數有:

signal.**SIG_DFL**

這是兩種標準訊號處理選項之一；它會簡單地執行訊號的預設功能。例如，在大多數系統上，SIGQUIT 的預設動作是轉儲 (dump) 核心🄴退出，而*SIGCHLD* 的預設動作是直接忽略。

signal.**SIG_IGN**

這是另一個標准的訊號處理程式，會直接忽略給定的訊號。

signal.**SIGABRT**

來自 *abort(3)* 的中止訊號。

signal.**SIGALRM**

來自 *alarm(2)* 的計時器訊號。

可用性: Unix.

signal.**SIGBREAK**

從鍵盤中斷 (CTRL + BREAK)。

可用性: Windows.

signal.**SIGBUS**

匯流排錯誤（記憶體存取不良）。

可用性: Unix.

signal.**SIGCHLD**

子行程停止或終止。

可用性: Unix.

signal.**SIGCLD**

SIGCHLD 的🄵名。

可用性: not macOS.

signal.**SIGCONT**

如果目前行程是被停止的，則繼續運行

可用性: Unix.

signal.**SIGFPE**

浮點運算例外。例如除以零。

> **也參考**
>
> *ZeroDivisionError* 會在除法或模運算 (modulo operation) 的第二個引數🄵零時引發。

signal.**SIGHUP**

偵測到控制終端機🄵斷 (hangup) 或控制行程死亡。

可用性: Unix.

signal.**SIGILL**

非法指令。

signal.**SIGINT**

從鍵盤中斷 (CTRL + C)。

預設動作是引發*KeyboardInterrupt*。

signal.**SIGKILL**

殺死訊號。

它無法被捕捉、阻擋或忽略。

可用性: Unix.

signal.**SIGPIPE**

管道中斷 (broken pipe)：寫到🄵有讀取器 (reader) 的管道。

預設動作是忽略訊號。

可用性: Unix.

signal.**SIGSEGV**

記憶體區段錯誤 (segmentation fault)：無效記憶體參照。

signal.**SIGSTKFLT**

輔助處理器 (coprocessor) 上的堆🄵錯誤 (stack fault)。Linux 核心不會引發此訊號：它只能在使用者空間 (user space) 中引發。

可用性: Linux.

在訊號可用的架構上。請參🄵 *signal(7)* 🄵上手🄵以取得更多資訊。

在 3.11 版被加入.

signal.**SIGTERM**

終止訊號。

signal.**SIGUSR1**

使用者定義訊號 1。

可用性: Unix.

signal.**SIGUSR2**

使用者定義訊號 2。

可用性: Unix.

signal.**SIGWINCH**

視窗調整大小訊號。

可用性: Unix.

SIG*

所有的訊號編號都是以符號定義的。例如，🄵斷訊號被定義🄵*signal.SIGHUP*；變數名稱與 C 程式中使用的名稱相同，可在 <signal.h> 中找到。Unix 🄵上手🄵'signal()' 列出了現有的訊號（在某些系統上是 *signal(2)*，在其他系統上是在 *signal(7)* 中）。請注意，🄵非所有系統都會定義同一套訊號名稱；只有那些由系統所定義的名稱才會由這個模組定義。

signal.**CTRL_C_EVENT**

與 Ctrl+C 擊鍵 (keystroke) 事件相對應的訊號。此訊號只能與*os.kill()* 搭配使用。

可用性: Windows.

在 3.2 版被加入.

signal.**CTRL_BREAK_EVENT**

與 Ctrl+Break 擊鍵事件相對應的訊號。此訊號只能與*os.kill()* 搭配使用。

可用性: Windows.

在 3.2 版被加入.

signal.**NSIG**

比最高編號訊號的編號多一。使用*valid_signals()* 來取得有效的訊號編號。

signal.**ITIMER_REAL**

即時🄵少間隔計時器 (interval timer)，🄵在到期時送出*SIGALRM*。

signal.**ITIMER_VIRTUAL**

僅在執行行程時遞🄵間隔計時器，🄵在到期時傳送 SIGVTALRM。

signal.**ITIMER_PROF**

當行程執行或系統代表行程執行時，都會🄵少間隔計時器。與 ITIMER_VIRTUAL 配合，這個計時器通常用來分析 (profile) 應用程式在使用者空間與核心空間 (kernel space) 所花費的時間。SIGPROF 會在到期時傳送。

signal.**SIG_BLOCK**

pthread_sigmask() 的 *how* 參數的可能值，表示訊號將被阻檔。

在 3.3 版被加入.

signal.**SIG_UNBLOCK**

pthread_sigmask() 的 *how* 參數的可能值，表示訊號將被解除阻檔。

在 3.3 版被加入.

signal.**SIG_SETMASK**

pthread_sigmask() 的 *how* 參數的可能值，表示訊號遮罩 (signal mask) 要被取代。

在 3.3 版被加入.

signal 模組定義了一個例外：

exception signal.**ItimerError**

當*setitimer()* 或*getitimer()* 底層實作發生錯誤時引發訊號。如果傳給*setitimer()* 的是無效的間隔計時器或負數時間，則預期會發生此錯誤。這個錯誤是*OSError* 的子型🄵。

在 3.3 版被加入: 此錯誤過去是*IOError* 的子型🄵，現在是*OSError* 的🄵名。

signal 模組定義了下列函式：

signal.**alarm**(*time*)

如果 *time* 非零，則此函式會要求在 *time* 秒後傳送*SIGALRM* 訊號給該行程。任何先前排程 (scheduled) 的警報都會被取消（任何時候都只能排程一個警報）。回傳值是先前設定的警報原本再等多久就會被傳送的秒數。如果 *time* 🄵零，則不會去排程任何警報，且已排程的警報會被取消。如果回傳值🄵零，則代表目前未排程任何警報。

可用性: Unix.

更多資訊請見 *alarm(2)* 🄵上手🄵。

signal.**getsignal**(*signalnum*)

回傳訊號 *signalnum* 的目前訊號處理程式。回傳值可以是一個可呼叫的 Python 物件，或是特殊值*signal.SIG_IGN*、*signal.SIG_DFL* 或*None* 之一。這🄵的*signal.SIG_IGN* 表示訊號先前被忽略，*signal.SIG_DFL* 表示訊號先前使用預設的處理方式，而 None 表示先前的訊號處理程式🄵未從 Python 安裝。

signal.**strsignal**(*signalnum*)

回傳訊號 *signalnum* 的描述，例如*SIGINT* 的"Interrupt"。如果 *signalnum* 🄵有描述，則回傳*None*。如果 *signalnum* 無效則會引發*ValueError*。

在 3.8 版被加入.

signal.**valid_signals**()

回傳此平台上的有效訊號編號集合。如果某些訊號被系統保留作🄵🄵部使用，則此值可能小於 range(1, NSIG)。

在 3.8 版被加入.

signal.**pause**()

使行程休眠，直到接收到訊號；然後呼叫適當的處理程式。不會回傳任何東西。

可用性: Unix.

更多資訊請見 *signal(2)* 🄵上手🄵。

也請見*sigwait()*、*sigwaitinfo()*、*sigtimedwait()* 和*sigpending()*。

signal.**raise_signal**(*signum*)

傳送訊號至呼叫的行程。不會回傳任何東西。

在 3.8 版被加入.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)

傳送訊號 *sig* 到檔案描述器 *pidfd* 所指的行程。Python 目前不支援 *siginfo* 參數；它必須是 None。*flags* 引數是提供給未來的擴充；目前🄵有定義旗標值。

更多資訊請見 *pidfd_send_signal(2)* 🄵上手🄵。

可用性: Linux >= 5.1, Android >= *build-time* API level 31

在 3.9 版被加入.

signal.**pthread_kill**(*thread_id*, *signalnum*)

將訊號 *signalnum* 傳送給與呼叫者在同一行程中的另一個執行緒 *thread_id*。目標執行緒能執行任何程式碼（無論🄵 Python 與否）。但是，如果目標執行緒正執行 Python 直譯器，Python 訊號處理程式將會由主直譯器的主執行緒來執行。因此，向特定 Python 執行緒發送訊號的唯一意義是🄵制執行中的系統呼叫以*InterruptedError* 方式失敗。

使用*threading.get_ident()* 或*threading.Thread* 物件的*ident* 屬性來取得 *thread_id* 的適當值。

如果 *signalnum* 🄵 0，則不會傳送訊號，但仍會執行錯誤檢查；這可用來檢查目標執行緒是否仍在執行。

引發一個附帶引數 thread_id、signalnum 的稽核事件 signal.pthread_kill。

可用性: Unix.

更多資訊請見 *pthread_kill(3)* 🄵上手🄵。

另請參🄵*os.kill()*。

在 3.3 版被加入.

signal.**pthread_sigmask**(*how*, *mask*)

🄵取和/或變更呼叫執行緒的訊號遮罩。訊號遮罩是目前阻擋呼叫者傳送的訊號集合。將舊的訊號遮罩作🄵一組訊號集合回傳。

呼叫的行🄵取🄴於 *how* 的值，如下所示。

- *SIG_BLOCK*：被阻檔的訊號集合是目前訊號集合與 *mask* 引數的聯集。
- *SIG_UNBLOCK*：將 *mask* 中的訊號從目前阻檔的訊號集合中移除。嘗試🄵未被阻檔的訊號解除阻檔是允許的。
- *SIG_SETMASK*：將被阻檔的訊號集合設定🄵 *mask* 引數。

mask 是一組訊號編號（例如 {*signal.SIGINT*, *signal.SIGTERM*}）的集合。使用*valid_signals()* 來取得包含所有訊號的完整遮罩。

例如，`signal.pthread_sigmask(signal.SIG_BLOCK, [])` 會讀取呼叫執行緒的訊號遮罩。

SIGKILL 和 SIGSTOP 不能被阻檔。

可用性: Unix.

更多資訊請見 *sigprocmask(2)* 與 *pthread_sigmask(3)* 🄵上手🄵。

另請參🄵*pause()*、*sigpending()* 與*sigwait()*。

在 3.3 版被加入.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)

設定由 *which* 指定的間隔計時器（*signal.ITIMER_REAL*、*signal.ITIMER_VIRTUAL* 或*signal.ITIMER_PROF* 之一）🄵在 *seconds*（接受浮點數，與*alarm()* 不同）之後🄵動，在之後的每 *interval* 秒🄵動一次（如果 *interval* 非零）。*which* 指定的間隔計時器可透過將 *seconds* 設定🄵零來清除它。

當間隔計時器🄵動時，一個訊號會被傳送給行程。傳送的訊號取🄵於使用的計時器；*signal.ITIMER_REAL* 會傳送*SIGALRM*，*signal.ITIMER_VIRTUAL* 會傳送 SIGVTALRM，而*signal.ITIMER_PROF* 會傳送 SIGPROF。

舊值會以一個元組回傳：(delay, interval)。

嘗試傳入無效的間隔計時器會導致*ItimerError*。

可用性: Unix.

signal.**getitimer**(*which*)

回傳由 *which* 指定之間隔計時器的目前值。

可用性: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)

設定🄵醒檔案描述器🄵 *fd*。當接收到其處理程式已有被🄵🄵的訊號時，訊號編號會以單一位元組寫入 fd。如果你🄵有🄵你在意的訊號🄴🄵處理程式，就不會寫入到🄵醒 fd。這可被函式庫用來🄵醒輪詢 (wakeup a poll) 或 select 呼叫，讓訊號得以完全處理。

回傳舊的🄵醒 fd（如果檔案描述器🄵醒未🄵用，則回傳 -1）。如果 *fd* 🄵 -1，則會停用檔案描述器🄵醒。如果不是 -1，*fd* 必須是非阻塞的。在再次呼叫輪詢或 select 之前，由函式庫來🄵定是否移除 *fd* 中的任何位元組。

當🄵用執行緒時，這個函式只能從主直譯器的主執行緒來呼叫；嘗試從其他執行緒呼叫它將會引發*ValueError* 例外。

使用這個函式有兩種常見的方式。在這兩種方法中，當訊號抵達時，你都要使用 fd 來🄵醒，但它們的不同之處在於如何判斷哪個或哪些訊號有抵達。

在第一種方法中，我們從 fd 的緩衝區中讀取資料，而位元組值則提供訊號編號。這個方法很簡單，但在少數情F下可能會遇到問題：一般來F，fd 的緩衝區空間有限，如果太多訊號來得太快，那F緩衝區可能會滿，而有些訊號可能會遺失。如果你使用這種方法，那F你應該設定 `warn_on_full_buffer=True`，這至少會在訊號F失時將警告印出到 stderr。

在第二種方法中，我們只會將F醒 fd 用於F醒，而忽略實際的位元組值。在這種情F下，我們只在乎 fd 的緩衝區是空或非空；即便緩衝區滿了也不代表有問題。如果你使用這種方法，那F你應該設定 `warn_on_full_buffer=False`，這樣你的使用者就不會被F假的警告訊息所混淆。

在 3.5 版的變更: 在 Windows 上，此功能現在也支援 socket 處理程式。

在 3.7 版的變更: 新增 `warn_on_full_buffer` 參數。

`signal.`**`siginterrupt`**`(`*signalnum*, *flag*`)`

改變系統呼叫重新F動的行F：如果 *flag* 是*`False`*，系統呼叫會在被訊號 *signalnum* 中斷時重新F動，否則系統呼叫會被中斷。不會回傳任何東西。

可用性: Unix.

更多資訊請見 *`siginterrupt(3)`* F上手F。

請注意，使用*`signal()`* 安裝訊號處理程式，會透過隱式呼叫 `siginterrupt()` 來將重新F動的行F重設F可中斷，且指定訊號的 *flag* 值F true。

`signal.`**`signal`**`(`*signalnum*, *handler*`)`

將訊號 *signalnum* 的處理程式設定F函式 *handler*。*handler* 可以是帶兩個引數的可呼叫 Python 物件（見下面），或是特殊值*`signal.SIG_IGN`* 或*`signal.SIG_DFL`* 之一。先前的訊號處理程式將會被回傳（請參F上面*`getsignal()`* 的F明）。（更多資訊請參F Unix F上手F *`signal(2)`*）。

當F用執行緒時，這個函式只能從主直譯器的主執行緒來呼叫；嘗試從其他執行緒呼叫它將會引發*`ValueError`* 例外。

handler 被呼叫時有兩個引數：訊號編號和目前的堆F frame（`None` 或一個 frame 物件；關於 frame 物件的描述，請參F型F階層中的描述或*`inspect`* 模組中的屬性描述）。

在 Windows 上，*`signal()`* 只能在使用*`SIGABRT`*、*`SIGFPE`*、*`SIGILL`*、*`SIGINT`*、*`SIGSEGV`*、*`SIGTERM`* 或*`SIGBREAK`* 時呼叫。在其他情F下會引發*`ValueError`*。請注意，F非所有系統都定義相同的訊號名稱；如果訊號名稱F有被定義F `SIG*` 模組層級常數，則會引發*`AttributeError`* 錯誤。

`signal.`**`sigpending`**`()`

檢查待傳送至呼叫執行緒的訊號集合（即阻檔時已被提出的訊號）。回傳待定訊號的集合。

可用性: Unix.

更多資訊請見 *`sigpending(2)`* F上手F。

另請參F*`pause()`*、*`pthread_sigmask()`* 與*`sigwait()`*。

在 3.3 版被加入.

`signal.`**`sigwait`**`(`*sigset*`)`

暫停呼叫執行緒的執行，直到送出訊號集合 *sigset* 中指定的一個訊號。函式接受訊號（將其從待定訊號清單中移除），F回傳訊號編號。

可用性: Unix.

更多資訊請見 *`sigwait(3)`* F上手F。

另也請見*`pause()`*、*`pthread_sigmask()`*、*`sigpending()`*、*`sigwaitinfo()`* 和*`sigtimedwait()`*。

在 3.3 版被加入.

`signal.`**`sigwaitinfo`**`(`*sigset*`)`

暫停呼叫執行緒的執行，直到送出訊號集合 *sigset* 中指定的一個訊號。該函式接受訊號，F將其從待定訊號清單中移除。如果 *sigset* 中的一個訊號已經是呼叫執行緒的待定訊號，函式會立即回傳該訊號的相關資訊。對於已傳送的訊號，訊號處理程式不會被呼叫。如果被不在 *sigset* 中的訊號中斷，函式會引發*`InterruptedError`*。

回傳值是一個物件，代表 `siginfo_t` 結構所包含的資料，即 `si_signo`、`si_code`、`si_errno`、`si_pid`、`si_uid`、`si_status`、`si_band`。

可用性: Unix.

更多資訊請見 *sigwaitinfo(2)* 🄵上手🄵。

另請參🄵*pause()*、*sigwait()* 與*sigtimedwait()*。

在 3.3 版被加入.

在 3.5 版的變更: 現在如果被不在 *sigset* 中的訊號中斷，且訊號處理程式🄵有引發例外，則會重試函式（理由請參🄵 **PEP 475**）。

signal.**sigtimedwait**(*sigset*, *timeout*)

類似*sigwaitinfo()*，但需要額外的 *timeout* 引數指定逾時時間。如果 *timeout* 指定🄵 `0`，會執行輪詢。如果發生逾時則會回傳*None*。

可用性: Unix.

更多資訊請見 *sigtimedwait(2)* 🄵上手🄵。

另請參🄵*pause()*、*sigwait()* 與*sigwaitinfo()*。

在 3.3 版被加入.

在 3.5 版的變更: 現在如果被不在 *sigset* 中的訊號中斷，且訊號處理程式🄵有引發例外，則會使用重新計算的 *timeout* 重試函式（理由請參🄵 **PEP 475**）。

19.6.3 范例

這是一個最小範例程式。它使用*alarm()* 函式來限制等待開🄵檔案的時間；如果檔案是用於可能未開🄵的序列裝置，這會很有用，因🄵這通常會導致*os.open()* 無限期地被擱置。解🄵方法是在開🄵檔案前設定一個 5 秒的警報；如果操作時間過長，警報訊號就會被送出，而處理程式會🄵生例外。

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({signum})')
    raise OSError("Couldn't open device!")

# 設定訊號處理程式與五秒警報
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# 這個 open() 可能無限期地被擱置
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # 停用警報
```

19.6.4 關於 SIGPIPE 的🄵明

將程式的輸出管道化到 *head(1)* 之類的工具，會在你的行程的標准輸出接收器提早關閉時，導致*SIGPIPE* 訊號傳送給你的行程。這會導致類似 `BrokenPipeError: [Errno 32] Broken pipe` 的例外。要處理這種情🄵，請將你的進入點包裝成如下的樣子來捕捉這個例外：

```
import os
import sys

def main():
    try:
        # 模擬大量輸出（你的程式取代此🄵圈）
        for x in range(10000):
```

(繼續下一頁)

(繼續上一頁)

```
            print("y")
        # 在這🄵清除輸出以🄵制 SIGPIPE 在這個 try 區塊
        # 中被觸發
        sys.stdout.flush()
    except BrokenPipeError:
        # Python 在退出時清除標准串流；🄵剩下的輸出重新導向
        # 至 devnull 來避免關閉時的 BrokenPipeError
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python 在 EPIPE 時以錯誤碼 1 退出

if __name__ == '__main__':
    main()
```

不要🄵了避免*BrokenPipeError* 而將*SIGPIPE* 之處置 (disposition) 設定🄵*SIG_DFL*。這樣做會導致你的程式在寫入任何 socket 連🄵時被中斷而意外退出。

19.6.5 訊號處理程式與例外的🄵明

如果訊號處理程式🄵生例外，例外會傳送到主執行緒🄵可能在任何*bytecode* 指令之後發生。最值得注意的是，*KeyboardInterrupt* 可能在執行過程中的任何時候出現。大多數 Python 程式碼，包括標准函式庫，都無法避免這種情🄵，因此*KeyboardInterrupt*（或任何其他由訊號處理程式🄵生的例外）可能會在罕見的情🄵下使程式處於預期之外的狀態。

🄵了🄵明這個問題，請參考以下程式碼：

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # 如果 KeyboardInterrupt 在此發生則一切正常
        self.lock.acquire()
        # 如果 KeyboardInterrupt 在此發生，__exit__ 將不會被呼叫
        ...
        # KeyboardInterrupt 可能在函式回傳之前發生

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

對許多程式來🄵，尤其是那些只想在*KeyboardInterrupt* 時退出的程式，這🄵不是問題，但是對於🄵雜或需要高可靠性的應用程式來🄵，應該避免從訊號處理程式🄵生例外。它們也應該避免將捕獲*KeyboardInterrupt* 作🄵一種優雅關閉 (gracefully shutting down) 的方式。相反地，它們應該安裝自己的*SIGINT* 處理程式。以下是 HTTP 伺服器避免*KeyboardInterrupt* 的範例：

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
```

(繼續下一頁)

(繼續上一頁)

```
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

19.7 mmap --- 記憶體對映檔案的支援

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

Memory-mapped file objects behave like both *bytearray* and like *file objects*. You can use mmap objects in most places where *bytearray* are expected; for example, you can use the *re* module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the *mmap* constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its *fileno()* method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the *os.open()* function, which returns a file descriptor directly (the file still needs to be closed when done).

> **備🄵**
>
> If you want to create a memory-mapping for a writable, buffered file, you should *flush()* the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of four values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively, or `ACCESS_DEFAULT` to defer to *prot*. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a *TypeError* exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

在 3.7 版的變更: 新增 `ACCESS_DEFAULT` 常數。

To map anonymous memory, -1 should be passed as the fileno along with the length.

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*, *offset=0*)

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is `0`, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened,

otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the *tagname* parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the `ALLOCATIONGRANULARITY`.

引發一個附帶引數 `fileno`、`length`、`access`、`offset` 的稽核事件 `mmap.__new__`。

class `mmap.`**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE | PROT_READ*, *access=ACCESS_DEFAULT*, *offset=0*, *, *trackfd=True*)

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when *mmap* is called.

flags specifies the nature of the mapping. *MAP_PRIVATE* creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and *MAP_SHARED* creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is *MAP_SHARED*. Some systems have additional possible flags with the full list specified in *MAP_* constants*.

prot, if specified, gives the desired memory protection; the two most useful values are `PROT_READ` and `PROT_WRITE`, to specify that the pages may be read or written. *prot* defaults to `PROT_READ | PROT_WRITE`.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of `ALLOCATIONGRANULARITY` which is equal to `PAGESIZE` on Unix systems.

If *trackfd* is `False`, the file descriptor specified by *fileno* will not be duplicated, and the resulting `mmap` object will not be associated with the map's underlying file. This means that the *size()* and *resize()* methods will fail. This mode is useful to limit the number of open file descriptors.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with the physical backing store on macOS.

在 3.13 版的變更: The *trackfd* parameter was added.

This example shows a simple way of using *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap can also be used as a context manager in a `with` statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

在 3.2 版被加入: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # 在子行程中
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

引發一個附帶引數 `fileno`、`length`、`access`、`offset` 的稽核事件 `mmap.__new__`。

Memory-mapped file objects support the following methods:

close()

Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed

`True` if the file is closed.

在 3.2 版被加入.

find(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

flush()

flush(*offset*, *size*, /)

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

`None` is returned to indicate success. An exception is raised when the call failed.

在 3.8 版的變更: Previously, a nonzero value was returned on success; zero was returned on error under Windows. A zero value was returned on success; an exception was raised on error under Unix.

madvise(*option*[, *start*[, *length*]])

Send advice *option* to the kernel about the memory region beginning at *start* and extending *length* bytes. *option* must be one of the *MADV_* constants* available on the system. If *start* and *length* are omitted, the entire mapping is spanned. On some systems (including Linux), *start* must be a multiple of the `PAGESIZE`.

Availability: Systems with the `madvise()` system call.

在 3.8 版被加入.

move(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with ACCESS_READ, then calls to move will raise a *TypeError* exception.

read([*n*])

Return a *bytes* containing up to *n* bytes starting from the current file position. If the argument is omitted, None or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

在 3.3 版的變更: Argument can be omitted or None.

read_byte()

Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()

Returns a single line, starting at the current file position and up to the next newline. The file position is updated to point after the bytes that were returned.

resize(*newsize*)

Resizes the map and the underlying file, if any.

Resizing a map created with *access* of ACCESS_READ or ACCESS_COPY, will raise a *TypeError* exception. Resizing a map created with *trackfd* set to False, will raise a *ValueError* exception.

On Windows: Resizing the map will raise an *OSError* if there are other maps against the same named file. Resizing an anonymous map (ie against the pagefile) will silently create a new map with the original data copied over up to the length of the new size.

在 3.11 版的變更: Correctly fails if attempting to resize when another map is held Allows resize against an anonymous map on Windows

rfind(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns -1 on failure.

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

seek(*pos*[, *whence*])

Set the file's current position. *whence* argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file's end).

在 3.13 版的變更: Return the new absolute position instead of None.

seekable()

Return whether the file supports seeking, and the return value is always True.

在 3.13 版被加入.

size()

Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()

Returns the current position of the file pointer.

write(*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than len(bytes), since if the write fails, a *ValueError* will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will raise a *TypeError* exception.

在 3.5 版的變更: Writable *bytes-like object* is now accepted.

在 3.6 版的變更: The number of bytes written is now returned.

write_byte(*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *`TypeError`* exception.

19.7.1 MADV_* 常數

mmap.**MADV_NORMAL**

mmap.**MADV_RANDOM**

mmap.**MADV_SEQUENTIAL**

mmap.**MADV_WILLNEED**

mmap.**MADV_DONTNEED**

mmap.**MADV_REMOVE**

mmap.**MADV_DONTFORK**

mmap.**MADV_DOFORK**

mmap.**MADV_HWPOISON**

mmap.**MADV_MERGEABLE**

mmap.**MADV_UNMERGEABLE**

mmap.**MADV_SOFT_OFFLINE**

mmap.**MADV_HUGEPAGE**

mmap.**MADV_NOHUGEPAGE**

mmap.**MADV_DONTDUMP**

mmap.**MADV_DODUMP**

mmap.**MADV_FREE**

mmap.**MADV_NOSYNC**

mmap.**MADV_AUTOSYNC**

mmap.**MADV_NOCORE**

mmap.**MADV_CORE**

mmap.**MADV_PROTECT**

mmap.**MADV_FREE_REUSABLE**

mmap.**MADV_FREE_REUSE**

These options can be passed to *`mmap.madvise()`*. Not every option will be present on every system.

Availability: Systems with the madvise() system call.

在 3.8 版被加入.

19.7.2 MAP_* 常數

mmap.**MAP_SHARED**

mmap.**MAP_PRIVATE**

mmap.**MAP_32BIT**

mmap.**MAP_ALIGNED_SUPER**

mmap.**MAP_ANON**

mmap.**MAP_ANONYMOUS**

mmap.**MAP_CONCEAL**

mmap.**MAP_DENYWRITE**

mmap.**MAP_EXECUTABLE**

mmap.**MAP_HASSEMAPHORE**

mmap.**MAP_JIT**

mmap.**MAP_NOCACHE**

mmap.**MAP_NOEXTEND**

mmap.**MAP_NORESERVE**

mmap.**MAP_POPULATE**

mmap.**MAP_RESILIENT_CODESIGN**

mmap.**MAP_RESILIENT_MEDIA**

mmap.**MAP_STACK**

mmap.**MAP_TPRO**

mmap.**MAP_TRANSLATED_ALLOW_EXECUTE**

mmap.**MAP_UNIX03**

These are the various flags that can be passed to *mmap.mmap()*. *MAP_ALIGNED_SUPER* is only available at FreeBSD and *MAP_CONCEAL* is only available at OpenBSD. Note that some options might not be present on some systems.

在 3.10 版的變更: 新增*MAP_POPULATE* 常數。

在 3.11 版被加入: 新增*MAP_STACK* 常數。

在 3.12 版被加入: 新增*MAP_ALIGNED_SUPER* 和*MAP_CONCEAL* 常數。

在 3.13 版被加入: Added *MAP_32BIT*, *MAP_HASSEMAPHORE*, *MAP_JIT*, *MAP_NOCACHE*, *MAP_NOEXTEND*, *MAP_NORESERVE*, *MAP_RESILIENT_CODESIGN*, *MAP_RESILIENT_MEDIA*, *MAP_TPRO*, *MAP_TRANSLATED_ALLOW_EXECUTE*, and *MAP_UNIX03* constants.

CHAPTER 20

網際網路資料處理

本章描述了支援網際網路上處理常用資料格式的模組。

20.1 `email` --- 郵件和 MIME 處理套件

原始碼： Lib/email/__init__.py

The *email* package is a library for managing email messages. It is specifically *not* designed to do any sending of email messages to SMTP (**RFC 2821**), NNTP, or other servers; those are functions of modules such as *smtplib*. The *email* package attempts to be as RFC-compliant as possible, supporting **RFC 5322** and **RFC 6532**, as well as such MIME-related RFCs as **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183**, and **RFC 2231**.

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components.

The central component of the package is an ”object model” that represents email messages. An application interacts with the package primarily through the object model interface defined in the *message* sub-module. The application can use this API to ask questions about an existing email, to construct a new email, or to add or remove email subcomponents that themselves use the same object model interface. That is, following the nature of email messages and their MIME subcomponents, the email object model is a tree structure of objects that all provide the *EmailMessage* API.

The other two major components of the package are the *parser* and the *generator*. The parser takes the serialized version of an email message (a stream of bytes) and converts it into a tree of *EmailMessage* objects. The generator takes an *EmailMessage* and turns it back into a serialized byte stream. (The parser and generator also handle streams of text characters, but this usage is discouraged as it is too easy to end up with messages that are not valid in one way or another.)

The control component is the *policy* module. Every *EmailMessage*, every *generator*, and every *parser* has an associated *policy* object that controls its behavior. Usually an application only needs to specify the policy when an *EmailMessage* is created, either by directly instantiating an *EmailMessage* to create a new email, or by parsing an input stream using a *parser*. But the policy can be changed when the message is serialized using a *generator*. This allows, for example, a generic email message to be parsed from disk, but to serialize it using standard SMTP settings when sending it to an email server.

The email package does its best to hide the details of the various governing RFCs from the application. Conceptually the application should be able to treat the email message as a structured tree of unicode text and binary attachments, without having to worry about how these are represented when serialized. In practice, however, it is often necessary

to be aware of at least some of the rules governing MIME messages and their structure, specifically the names and nature of the MIME "content types" and how they identify multipart documents. For the most part this knowledge should only be required for more complex applications, and even then it should only be the high level structure in question, and not the details of how those structures are represented. Since MIME content types are used widely in modern internet software (not just email), this will be a familiar concept to many programmers.

The following sections describe the functionality of the `email` package. We start with the `message` object model, which is the primary interface an application will use, and follow that with the `parser` and `generator` components. Then we cover the `policy` controls, which completes the treatment of the main components of the library.

The next three sections cover the exceptions the package may raise and the defects (non-compliance with the RFCs) that the `parser` may detect. Then we cover the `headerregistry` and the `contentmanager` sub-components, which provide tools for doing more detailed manipulation of headers and payloads, respectively. Both of these components contain features relevant to consuming and producing non-trivial messages, but also document their extensibility APIs, which will be of interest to advanced applications.

Following those is a set of examples of using the fundamental parts of the APIs covered in the preceding sections.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the `Message` class, cover the legacy `compat32` API that deals much more directly with the details of how email messages are represented. The `compat32` API does *not* hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the `compat32` API for backward compatibility reasons.

在 3.6 版的變更: Docs reorganized and rewritten to promote the new `EmailMessage`/`EmailPolicy` API.

Contents of the `email` package documentation:

20.1.1 `email.message`：表示電子郵件訊息

原始碼： Lib/email/message.py

在 3.6 版被加入:[1]

The central class in the `email` package is the `EmailMessage` class, imported from the `email.message` module. It is the base class for the `email` object model. `EmailMessage` provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as `multipart/*` or `message/rfc822`.

The conceptual model provided by an `EmailMessage` object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** body of the message, which might be a list of sub-`EmailMessage` objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The `EmailMessage` dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. The keys are ordered, but unlike a real dict, there can be duplicates. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of `EmailMessage` objects, for MIME container documents such as `multipart/*` and `message/rfc822` message objects.

[1] Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message: Representing an email message using the compat32 API*.

class email.message.**EmailMessage**(*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the *max_line_length* of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as "7 bit clean" when *utf8* is `False`, which is the default.

在 3.6 版的變更: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

在 3.4 版的變更: the method was changed to use `utf8=True`, thus producing an **RFC 6531**-like message representation, instead of being a direct alias for *as_string()*.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

__bytes__()

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()

Return `True` if the message's payload is a list of sub-*EmailMessage* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *EmailMessage* is of type `message/rfc822`.

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string. (See *mboxMessage* for a brief description of this header.)

get_unixfrom()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in an *EmailMessage* object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant headers named *name*.

Using the standard (non-`compat32`) policies, the returned value is an instance of a subclass of *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the *policy* defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()

Return a list of all the message's header field names.

values()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

以下是個範例：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a *KeyError*.

get_content_type()

Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by *get_default_type()*. If the *Content-Type* header is invalid, return `text/plain`.

(According to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value. **RFC 2045** defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.)

get_content_maintype()

Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()

Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the get_content_type methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is Content-Type (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the utf8 *charset* and None for the *language*.

If *replace* is False (the default) the header is moved to the end of the list of headers. If *replace* is True, the header will be updated in place.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the *params* attribute of the header value (for example, msg['Content-Type'].params['charset']).

在 3.4 版的變更: replace keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

get_filename(*failobj=None*)

Return the value of the filename parameter of the *Content-Disposition* header of the message. If the header does not have a filename parameter, this method falls back to looking for the name parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)

Return the value of the boundary parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)

Set the boundary parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)

Return the charset parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no charset parameter, *failobj* is returned.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()

Return `True` if there is a *Content-Disposition* header and its (case insensitive) value is `attachment`, `False` otherwise.

在 3.4.2 版的變更: is_attachment is now a method instead of a property, for consistency with *is_multipart()*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

在 3.5 版被加入.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()

The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

get_body(*preferencelist=('related', 'html', 'plain')*)

Return the MIME part that is the best candidate to be the "body" of the message.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *`Content-ID`* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *`Content-Disposition`* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *`Content-Type`* or whose *`Content-Type`* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()

Return an iterator over all of the immediate sub-parts of the message that are not candidate "body" parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *`Content-Disposition: attachment`*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *`Content-ID`* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

iter_parts()

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-`multipart`. (See also *`walk()`*.)

get_content(**args*, *content_manager=None*, ***kw*)

Call the *`get_content()`* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current *`policy`*.

set_content(**args*, *content_manager=None*, ***kw*)

Call the *`set_content()`* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current *`policy`*.

make_related(*boundary=None*)

Convert a non-`multipart` message into a `multipart/related` message, moving any existing *`Content-`* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(*boundary=None*)

Convert a non-`multipart` or a `multipart/related` into a `multipart/alternative`, moving any

existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)

Convert a non-`multipart`, a `multipart/related`, or a `multipart-alternative` into a `multipart/mixed`, moving any existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/related`, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the `multipart`. If the message is a non-`multipart`, call *make_related()* and then proceed as above. If the message is any other type of `multipart`, raise a *TypeError*. If *content_manager* is not specified, use the `content_manager` specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value `inline`.

add_alternative(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/alternative`, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the `multipart`. If the message is a non-`multipart` or `multipart/related`, call *make_alternative()* and then proceed as above. If the message is any other type of `multipart`, raise a *TypeError*. If *content_manager* is not specified, use the `content_manager` specified by the current *policy*.

add_attachment(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/mixed`, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the `multipart`. If the message is a non-`multipart`, `multipart/related`, or `multipart/alternative`, call *make_mixed()* and then proceed as above. If *content_manager* is not specified, use the `content_manager` specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value `attachment`. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and `inline` attachments (*Content-Disposition: inline*), by passing appropriate options to the `content_manager`.

clear()

Remove the payload and all of the headers.

clear_content()

Remove the payload and all of the *!Content-* headers, leaving all other headers intact and in their original order.

EmailMessage objects have the following instance attributes:

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be `None`.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

class email.message.**MIMEPart**(*policy=default*)

This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

🄵解

20.1.2 email.parser：剖析電子郵件訊息

原始碼： Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an *EmailMessage* object, adding headers using the dictionary interface, and adding payload(s) using *set_content()* and related methods, or they can be created by parsing a serialized representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root *EmailMessage* instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its *is_multipart()* method, and the subparts can be accessed via the payload manipulation methods, such as *get_body()*, *iter_parts()*, and *walk()*.

There are actually two parser interfaces available for use, the *Parser* API and the incremental *FeedParser* API. The *Parser* API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the *email* package's bundled parser and the *EmailMessage* class is embodied in the *Policy* class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `Policy` methods.

FeedParser API

The *BytesFeedParser*, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The *BytesFeedParser* can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the *BytesParser* API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The *BytesFeedParser*'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The *BytesFeedParser* is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's *defects* attribute with a list of any problems it found in a message. See the *email.errors* module for the list of defects that it can find.

Here is the API for the *BytesFeedParser*:

class email.parser.**BytesFeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Create a *BytesFeedParser* instance. Optional *_factory* is a no-argument callable; if not specified use the *message_factory* from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides *Message* as the default factory. All other policies provide *EmailMessage* as the default *_factory*. For more information on what else *policy* controls, see the *policy* documentation.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

在 3.2 版被加入.

在 3.3 版的變更: 新增 *policy* 關鍵字。

在 3.6 版的變更: *_factory* defaults to the policy `message_factory`.

feed(*data*)

Feed the parser some more data. *data* should be a *bytes-like object* containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if *feed()* is called after this method has been called.

class email.parser.**FeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Works like *BytesFeedParser* except that the input to the *feed()* method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if *utf8* is `True`, no binary attachments.

在 3.3 版的變更: 新增 *policy* 關鍵字。

Parser API

The *BytesParser* class, imported from the *email.parser* module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The *email.parser* module also provides *Parser* for parsing strings, and header-only parsers, *BytesHeaderParser* and *HeaderParser*, which can be used if you're only interested in the headers of the message. *BytesHeaderParser* and *HeaderParser* can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class email.parser.**BytesParser**(*_class=None*, *, *policy=policy.compat32*)

Create a *BytesParser* instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of *BytesFeedParser*.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

在 3.3 版的變更: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

在 3.6 版的變更: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the *readline()* and the `read()` methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if *utf8* is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a *Content-Transfer-Encoding* of `8bit`).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

parsebytes(*bytes*, *headersonly=False*)

Similar to the *parse()* method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a *BytesIO* instance first and calling *parse()*.

Optional *headersonly* is as with the *parse()* method.

在 3.2 版被加入.

class email.parser.**BytesHeaderParser**(*_class=None*, *, *policy=policy.compat32*)

Exactly like *BytesParser*, except that *headersonly* defaults to True.

在 3.3 版被加入.

class email.parser.**Parser**(*_class=None*, *, *policy=policy.compat32*)

This class is parallel to *BytesParser*, but handles string input.

在 3.3 版的變更: Removed the *strict* argument. Added the *policy* keyword.

在 3.6 版的變更: *_class* defaults to the policy message_factory.

parse(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the *readline()* and the *read()* methods on file-like objects.

Other than the text mode requirement, this method operates like *BytesParser.parse()*.

parsestr(*text*, *headersonly=False*)

Similar to the *parse()* method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a *StringIO* instance first and calling *parse()*.

Optional *headersonly* is as with the *parse()* method.

class email.parser.**HeaderParser**(*_class=None*, *, *policy=policy.compat32*)

Exactly like *Parser*, except that *headersonly* defaults to True.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level *email* package namespace.

email.**message_from_bytes**(*s*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure from a *bytes-like object*. This is equivalent to BytesParser().parsebytes(s). Optional *_class* and *policy* are interpreted as with the *BytesParser* class constructor.

在 3.2 版被加入.

在 3.3 版的變更: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_binary_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure tree from an open binary *file object*. This is equivalent to BytesParser().parse(fp). *_class* and *policy* are interpreted as with the *BytesParser* class constructor.

在 3.2 版被加入.

在 3.3 版的變更: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_string**(*s*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure from a string. This is equivalent to Parser().parsestr(s). *_class* and *policy* are interpreted as with the *Parser* class constructor.

在 3.3 版的變更: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure tree from an open *file object*. This is equivalent to Parser().parse(fp). *_class* and *policy* are interpreted as with the *Parser* class constructor.

在 3.3 版的變更: Removed the *strict* argument. Added the *policy* keyword.

在 3.6 版的變更: *_class* defaults to the policy message_factory.

Here's an example of how you might use *message_from_bytes()* at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for *is_multipart()*, and *iter_parts()* will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for *is_multipart()*, and *iter_parts()* will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their *is_multipart()* method will return `True`. The single element yielded by *iter_parts()* will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their *is_multipart()* method may return `False`. If such messages were parsed with the *FeedParser*, they will have an instance of the *MultipartInvariantViolationDefect* class in their *defects* attribute list. See *email.errors* for details.

20.1.3 `email.generator`: 產生 MIME 文件

原始碼： Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via *smtplib.SMTP.sendmail()*, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the *email.parser* module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming *policy* is used for both. That is, parsing the serialized byte stream via the *BytesParser* class and then regenerating the serialized byte stream using *BytesGenerator* should produce output identical to the input[1]. (On the other hand, using the generator on an *EmailMessage* constructed by program may result in changes to the *EmailMessage* object as defaults are filled in.)

The *Generator* class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not "8 bit clean".

To accommodate reproducible processing of SMIME-signed messages *Generator* disables header folding for message parts of type `multipart/signed` and all subparts.

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a *BytesGenerator* object that will write any message provided to the *flatten()* method, or any surrogateescape encoded text provided to the *write()* method, to the *file-like object outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

[1] This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no *email.policy* settings calling for automatic adjustments (for example, *refold_source* must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the *Message* or *EmailMessage* object passed to `flatten` to control the message generation. See *email.policy* for details on what *policy* controls.

在 3.2 版被加入.

在 3.3 版的變更: 新增關鍵字 *policy*。

在 3.6 版的變更: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the *BytesGenerator* instance was created.

If the *policy* option *cte_type* is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII *Content-Transfer-Encoding* of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see *mailbox*) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)

Return an independent clone of this *BytesGenerator* instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)

Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the *BytesGenerator*'s constructor.

As a convenience, *EmailMessage* provides the methods *as_bytes()* and `bytes(aMessage)` (a.k.a. *__bytes__()*), which simplify the generation of a serialized binary representation of a message object. For more detail, see *email.message*.

Because strings cannot represent binary data, the *Generator* class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible *Content-Transfer_Encoding*. Using the terminology of the email RFCs, you can think of this as *Generator* serializing to an I/O stream that is not "8 bit clean". In other words, most applications will want to be using *BytesGenerator*, and not *Generator*.

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a *Generator* object that will write any message provided to the *flatten()* method, or any text provided to the *write()* method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a `>` character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the *Message* or *EmailMessage* object passed to `flatten` to control the message generation. See *email.policy* for details on what *policy* controls.

在 3.3 版的變更: 新增關鍵字 *policy*。

在 3.6 版的變更: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the *Generator* instance was created.

If the *policy* option *cte_type* is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see *mailbox*) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

在 3.2 版的變更: Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone(*fp*)

Return an independent clone of this *Generator* instance with the exact same options, and *fp* as the new *outfp*.

write(*s*)

Write *s* to the *write* method of the *outfp* passed to the *Generator*'s constructor. This provides just enough file-like API for *Generator* instances to be used in the *print()* function.

As a convenience, *EmailMessage* provides the methods *as_string()* and `str(aMessage)` (a.k.a. *__str__()*), which simplify the generation of a formatted string representation of a message object. For more detail, see *email.message*.

The *email.generator* module also provides a derived class, *DecodedGenerator*, which is like the *Generator* base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

Act like *Generator*, except that for any subpart of the message passed to *Generator.flatten()*, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` -- Full MIME type of the non-*text* part
- `maintype` -- Main MIME type of the non-*text* part
- `subtype` -- Sub-MIME type of the non-*text* part
- `filename` -- Filename of the non-*text* part

- `description` -- Description associated with the non-*`text`* part
- `encoding` -- Content transfer encoding of the non-*`text`* part

If *fmt* is `None`, use the following default *fmt*:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

Optional *_mangle_from_* and *maxheaderlen* are as with the *`Generator`* base class.

🄵解

20.1.4 `email.policy`: Policy Objects

在 3.3 版被加入.

原始碼： Lib/email/policy.py

The *`email`* package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A *`Policy`* object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. *`Policy`* instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the *`parser`* classes and the related convenience functions, and for the *`Message`* class, this is the *`Compat32`* policy, via its corresponding pre-defined instance *`compat32`*. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to *`EmailMessage`* is the *`EmailPolicy`* policy, via its pre-defined instance *`default`*.

When a *`Message`* or *`EmailMessage`* object is created, it acquires a policy. If the message is created by a *`parser`*, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a *`generator`*, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the *`email.parser`* classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the *`parser`* module.

The first part of this documentation covers the features of *`Policy`*, an *abstract base class* that defines the features that are common to all policy objects, including *`compat32`*. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes *`EmailPolicy`* and *`Compat32`*, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

`Policy` instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new *`Policy`* instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
...
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of `0` or *None* indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see *`fold_binary()`* and *`utf8`* below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

raise_on_defect

If *`True`*, any defects encountered will be raised as errors. If *`False`* (the default), defects will be passed to the *`register_defect()`* method.

mangle_from_

If *`True`*, lines starting with *"From "* in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default: *`False`*.

在 3.5 版被加入.

message_factory

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case *`Message`* is used.

在 3.6 版被加入.

verify_generated_headers

If `True` (the default), the generator will raise *`HeaderWriteError`* instead of writing a header that is improperly folded or delimited, such that it would be parsed as multiple headers or joined with adjacent data. Such headers can be generated by custom header classes or bugs in the `email` module.

As it's a security feature, this defaults to `True` even in the *`Compat32`* policy. For backwards compatible, but unsafe, behavior, it must be set to `False` explicitly.

在 3.13 版被加入.

The following *`Policy`* method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)

Return a new *`Policy`* instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining *`Policy`* methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of *`MessageDefect`*.

The default implementation checks the *`raise_on_defect`* flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to *`register_defect()`*.

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of *`MessageDefect`*.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls *`handle_defect`*, *obj* will normally have a `defects` attribute that has an `append` method.

Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)

Return the maximum allowed number of headers named *name*.

Called when a header is added to an *EmailMessage* or *Message* object. If the returned value is not `0` or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a *ValueError* is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the ':' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(*name*, *value*)

The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting *linesep* characters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)

The same as *fold()*, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.**EmailPolicy**(***kw*)

This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the *message_factory* attribute is *EmailMessage*.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

在 3.6 版被加入:[1]

utf8

If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source

If the value for a header in the `Message` object originated from a *parser* (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	all values are refolded.

預設🄵 `long`。

header_factory

A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see *headerregistry*) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager

An object with at least two methods: get_content and set_content. When the *get_content()* or *set_content()* method of an *EmailMessage* object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to *raw_data_manager*.

在 3.4 版被加入.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_max_count(*name*)

Returns the value of the *max_count* attribute of the specialized class used to represent the header with the given name.

[1] Originally added in 3.3 as a *provisional feature*.

header_source_parse(*sourcelines*)

The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

header_fetch_parse(*name*, *value*)

If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)

Header folding is controlled by the *refold_source* policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using *splitlines()*. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

The same as *fold()* if *cte_type* is `7bit`, except that the returned value is bytes.

If *cte_type* is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

`email.policy.`**default**

An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

`email.policy.`**SMTP**

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

`email.policy.`**SMTPUTF8**

The same as `SMTP` except that *utf8* is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the *smtplib.SMTP.send_message()* method handles this automatically).

`email.policy.`**HTTP**

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

`email.policy.`**strict**

Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in *headerregistry*.

class email.policy.**Compat32**(***kw*)

This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, *compat32*, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the *Policy* default:

mangle_from_

The default is `True`.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_source_parse(*sourcelines*)

The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name and value are returned unmodified.

header_fetch_parse(*name*, *value*)

If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

email.policy.**compat32**

An instance of *Compat32*, providing backward compatibility with the behavior of the email package in Python 3.2.

🄵解

20.1.5 email.errors：例外和缺陷類🄵

原始碼： Lib/email/errors.py

The following exception classes are defined in the *email.errors* module:

exception email.errors.**MessageError**

This is the base class for all exceptions that the *email* package can raise. It is derived from the standard *Exception* class and defines no additional methods.

exception email.errors.**MessageParseError**

This is the base class for exceptions raised by the *Parser* class. It is derived from *MessageError*. This class is also used internally by the parser used by *headerregistry*.

exception email.errors.**HeaderParseError**

Raised under some error conditions when parsing the **RFC 5322** headers of a message, this class is derived from *MessageParseError*. The *set_boundary()* method will raise this error if the content type is unknown when the method is called. *Header* may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception email.errors.**BoundaryError**

Deprecated and no longer used.

exception email.errors.**MultipartConversionError**

Raised if the *attach()* method is called on an instance of a class derived from *MIMENonMultipart* (e.g. *MIMEImage*). *MultipartConversionError* multiply inherits from *MessageError* and the built-in *TypeError*.

exception email.errors.**HeaderWriteError**

Raised when an error occurs when the *generator* outputs headers.

exception email.errors.**MessageDefect**

This is the base class for all defects found when parsing email messages. It is derived from *ValueError*.

exception email.errors.**HeaderDefect**

This is the base class for all defects found when parsing email headers. It is derived from *MessageDefect*.

Here is the list of the defects that the *FeedParser* can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from *email.errors.MessageDefect*.

exception email.errors.**NoBoundaryInMultipartDefect**

A message claimed to be a multipart, but had no *boundary* parameter.

exception email.errors.**StartBoundaryNotFoundDefect**

The start boundary claimed in the *Content-Type* header was never found.

exception email.errors.**CloseBoundaryNotFoundDefect**

A start boundary was found, but no corresponding close boundary was ever found.

在 3.3 版被加入.

exception email.errors.**FirstHeaderLineIsContinuationDefect**

The message had a continuation line as its first header line.

exception email.errors.**MisplacedEnvelopeHeaderDefect**

A "Unix From" header was found in the middle of a header block.

exception email.errors.**MissingHeaderBodySeparatorDefect**

A line was found while parsing headers that had no leading white space but contained no ':'. Parsing continues assuming that the line represents the first line of the body.

在 3.3 版被加入.

exception email.errors.**MalformedHeaderDefect**

A header was found that was missing a colon, or was otherwise malformed.

在 3.3 版之後被🄵用: This defect has not been used for several Python versions.

exception email.errors.**MultipartInvariantViolationDefect**

A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its *is_multipart()* method may return False even though its content type claims to be *multipart*.

exception email.errors.**InvalidBase64PaddingDefect**

When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.

exception email.errors.**InvalidBase64CharactersDefect**

When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.

exception email.errors.**InvalidBase64LengthDefect**

When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.

exception email.errors.**InvalidDateDefect**

When decoding an invalid or unparsable date field. The original value is kept as-is.

20.1.6 email.headerregistry: 自訂標頭物件

原始碼: Lib/email/headerregistry.py

在 3.6 版被加入:[1]

Headers are represented by customized subclasses of *str*. The particular class used to represent a given header is determined by the *header_factory* of the *policy* in effect when the headers are created. This section documents the particular header_factory implemented by the email package for handling **RFC 5322** compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from *EmailPolicy*, all headers are produced by *HeaderRegistry* and have *BaseHeader* as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class *UnstructuredHeader* as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the *HeaderRegistry*. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of *HeaderRegistry*, and finally the support classes used to represent the data parsed from structured headers.

class email.headerregistry.**BaseHeader**(*name*, *value*)

name and *value* are passed to BaseHeader from the *header_factory* call. The string value of any header object is the *value* fully decoded to unicode.

這個基底類🄵定義了以下唯讀屬性:

[1] Originally added in 3.3 as a *provisional module*

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the *header_factory* call for *name*; that is, case is preserved.

defects

A tuple of *HeaderDefect* instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the *errors* module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same `name`. A value of `None` means unlimited. The `BaseHeader` value for this attribute is `None`; it is expected that specialized header classes will override this value as needed.

`BaseHeader` also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, *policy*)

Return a string containing *linesep* characters as required to correctly fold the header according to *policy*. A *cte_type* of `8bit` will be treated as if it were `7bit`, since headers may not contain arbitrary binary data. If *utf8* is `False`, non-ASCII data will be **RFC 2047** encoded.

`BaseHeader` by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, `BaseHeader` requires that the specialized class provide a *classmethod()* named `parse`. This method is called as follows:

```
parse(string, kwds)
```

`kwds` is a dictionary containing one pre-initialized key, `defects`. `defects` is an empty list. The parse method should append any detected defects to this list. On return, the `kwds` dictionary *must* contain values for at least the keys `decoded` and `defects`. `decoded` should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

`BaseHeader`'s `__new__` then creates the header instance, and calls its `init` method. The specialized class only needs to provide an `init` method if it wishes to set additional attributes beyond those provided by `BaseHeader` itself. Such an `init` method should look like this:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the `kwds` dictionary should be removed and handled, and the remaining contents of `kw` (and `args`) passed to the `BaseHeader` `init` method.

class email.headerregistry.**UnstructuredHeader**

An "unstructured" header is the default type of header in **RFC 5322**. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322**, an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047**, however, has an **RFC 5322** compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words into unicode, following the **RFC 2047** rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.**DateHeader**

RFC 5322 specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found "in the wild".

This header type provides the following additional attributes:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including `+0000`), then *datetime* will contain an aware `datetime` that uses *datetime.timezone* to record the timezone offset.

The `decoded` value of the header is determined by formatting the `datetime` according to the **RFC 5322** rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, *value* may be *datetime* instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the *localtime()* function from the *utils* module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.**AddressHeader**

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups

A tuple of *Group* objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose *display_name* is `None`.

addresses

A tuple of *Address* objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is "flattened" into a one dimensional list).

The `decoded` value of the header will have all encoded words decoded to unicode. *idna* encoded domain names are also decoded to unicode. The `decoded` value is set by *joining* the *str* value of the elements of the `groups` attribute with `', '`.

A list of *Address* and *Group* objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

class email.headerregistry.**SingleAddressHeader**

A subclass of *AddressHeader* that adds one additional attribute:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *policy*), accessing this attribute will result in a *ValueError*.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, *max_count* is set to 1.

class email.headerregistry.**MIMEVersionHeader**

There is really only one valid value for the *MIME-Version* header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-`None` values for the following attributes:

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

class email.headerregistry.**ParameterizedMIMEHeader**

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.**ContentTypeHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Type* header.

content_type

The content type string, in the form `maintype/subtype`.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

A *ParameterizedMIMEHeader* class that handles the *Content-Disposition* header.

content_disposition

`inline` and `attachment` are the only valid values in common use.

class email.headerregistry.**ContentTransferEncoding**

Handles the *Content-Transfer-Encoding* header.

cte

Valid values are `7bit`, `8bit`, `base64`, and `quoted-printable`. See **RFC 2045** for more information.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

This is the factory used by *EmailPolicy* by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

The default mappings are:

subject
UniqueUnstructuredHeader

date
UniqueDateHeader

resent-date
DateHeader

orig-date
UniqueDateHeader

sender
UniqueSingleAddressHeader

resent-sender
SingleAddressHeader

to
UniqueAddressHeader

resent-to
AddressHeader

cc
UniqueAddressHeader

resent-cc
AddressHeader

bcc
UniqueAddressHeader

resent-bcc
AddressHeader

from
UniqueAddressHeader

resent-from
AddressHeader

reply-to
UniqueAddressHeader

mime-version
MIMEVersionHeader

content-type
ContentTypeHeader

content-disposition
ContentDispositionHeader

content-transfer-encoding
ContentTransferEncodingHeader

message-id
MessageIDHeader

`HeaderRegistry` 有以下方法：

map_to_type(*self*, *name*, *cls*)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)

Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

或是:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322**.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

The `username` portion of the address, with all quoting removed.

domain

The `domain` portion of the address.

addr_spec

The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()

The `str` value of the object is the address quoted according to **RFC 5322** rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321**), `Address` handles one special case: if `username` and `domain` are both the empty string (or `None`), then the string value of the `Address` is `<>`.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a `Group` may also be used to represent single addresses that are not part of a group by setting *display_name* to `None` and providing a list of the single address as *addresses*.

display_name

The `display_name` of the group. If it is `None` and there is exactly one `Address` in `addresses`, then the `Group` represents a single address that is not in a group.

addresses

A possibly empty tuple of *Address* objects representing the addresses in the group.

__str__()

The `str` value of a `Group` is formatted according to **RFC 5322**, but with no Content Transfer Encoding of any non-ASCII characters. If `display_name` is none and there is a single `Address` in the `addresses` list, the `str` value will be the same as the `str` of that single `Address`.

🄵解

20.1.7 email.contentmanager：管理 MIME 🄵容

原始碼： Lib/email/contentmanager.py

在 3.6 版被加入:[1]

class email.contentmanager.**ContentManager**

Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the `get_content` and `set_content` dispatch methods.

get_content(*msg*, **args*, ***kw*)

Look up a handler function based on the `mimetype` of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found:

- the string representing the full MIME type (`maintype/subtype`)
- the string representing the `maintype`
- the empty string

If none of these keys produce a handler, raise a *KeyError* for the full MIME type.

set_content(*msg*, *obj*, **args*, ***kw*)

If the `maintype` is `multipart`, raise a *TypeError*; otherwise look up a handler function based on the type of *obj* (see next paragraph), call *clear_content()* on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found:

- the type itself (`typ`)
- the type's fully qualified name (`typ.__module__ + '.' + typ.__qualname__`).
- the type's `qualname` (`typ.__qualname__`)
- the type's `name` (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a *KeyError* for the fully qualified name of the type.

Also add a *MIME-Version* header if one is not present (see also *MIMEPart*).

add_get_handler(*key*, *handler*)

Record the function *handler* as the handler for *key*. For the possible values of *key*, see *get_content()*.

add_set_handler(*typekey*, *handler*)

Record *handler* as the function to call when an object of a type matching *typekey* is passed to *set_content()*. For the possible values of *typekey*, see *set_content()*.

[1] Originally added in 3.4 as a *provisional module*

Content Manager Instances

Currently the email package provides only one concrete content manager, *raw_data_manager*, although more may be added in the future. *raw_data_manager* is the *content_manager* provided by *EmailPolicy* and its derivatives.

email.contentmanager.**raw_data_manager**

This content manager provides only a minimum interface beyond that provided by *Message* itself: it deals only with text, raw byte strings, and *Message* objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

email.contentmanager.**get_content**(*msg*, *errors='replace'*)

Return the payload of the part as either a string (for `text` parts), an *EmailMessage* object (for `message/rfc822` parts), or a `bytes` object (for all other non-multipart types). Raise a *KeyError* if called on a `multipart`. If the part is a `text` part and *errors* is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

Add headers and payload to *msg*:

Add a *Content-Type* header with a `maintype/subtype` value.

- For `str`, set the MIME `maintype` to `text`, and set the subtype to *subtype* if it is specified, or `plain` if it is not.
- For `bytes`, use the specified *maintype* and *subtype*, or raise a *TypeError* if they are not specified.
- For *EmailMessage* objects, set the maintype to `message`, and set the subtype to *subtype* if it is specified or `rfc822` if it is not. If *subtype* is `partial`, raise an error (`bytes` objects must be used to construct `message/partial` parts).

If *charset* is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified *charset* is a known alias for a standard MIME charset name, use the standard charset instead.

If *cte* is set, encode the payload using the specified content transfer encoding, and set the *Content-Transfer-Encoding* header to that value. Possible values for *cte* are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a *cte* of `7bit` for an input that contains non-ASCII values), raise a *ValueError*.

- For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding. Prior to encoding, *str.splitlines()* is used to normalize all line boundaries, ensuring that each line of the payload is terminated by the current policy's *linesep* property (even if the original string did not end with one).
- For `bytes` objects, *cte* is taken to be base64 if not set, and the aforementioned newline translation is not performed.
- For *EmailMessage*, per **RFC 2046**, raise an error if a *cte* of `quoted-printable` or `base64` is requested for *subtype* `rfc822`, and for any *cte* other than `7bit` for *subtype* `external-body`. For `message/rfc822`, use `8bit` if *cte* is not specified. For all other values of *subtype*, use `7bit`.

> **備🄵**
>
> A *cte* of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but *`BytesGenerator`* does not serialize it correctly.

If *disposition* is set, use it as the value of the *`Content-Disposition`* header. If not specified, and *filename* is specified, add the header with the value `attachment`. If *disposition* is not specified and *filename* is also not specified, do not add the header. The only valid values for *disposition* are `attachment` and `inline`.

If *filename* is specified, use it as the value of the `filename` parameter of the *`Content-Disposition`* header.

If *cid* is specified, add a *`Content-ID`* header with *cid* as its value.

If *params* is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the *`Content-Type`* header.

If *headers* is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects (distinguished from strings by having a `name` attribute), add the headers to *msg*.

🄵解

20.1.8 `email`：范例

Here are a few examples of how to use the *`email`* package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Parsing **RFC 822** headers can easily be done by the using the classes from the *`parser`* module:

```
# Import the email modules we'll need
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
```

(繼續下一頁)

(繼續上一頁)

```
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function.
import smtplib

# Here are the email package modules we'll need.
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  You can also omit the subtype
# if you want MIMEImage to guess it.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Here's an example of how to send the entire contents of a directory as an email message:[1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
```

(繼續下一頁)

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

(繼續上一頁)

```
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
```

(繼續下一頁)

(繼續上一頁)

```
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

Here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == '__main__':
    main()
```

Here's an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recette
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}">
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))
```

(繼續下一頁)

(繼續上一頁)

```
# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """Return safety-sanitized html linked to partfiles.

    Rewrite the href="cid:...." attributes to point to the filenames in partfiles.
    Though not trivial, this should be possible using html.parser.
    """
    raise NotImplementedError("Add the magic needed")


# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
```

(繼續下一頁)

(繼續上一頁)

```
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

Up to the prompt, the output from the above is:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

🄵解

Legacy API:

20.1.9 `email.message.Message`: Representing an email message using the `compat32` API

The *`Message`* class is very similar to the *`EmailMessage`* class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the *`EmailMessage`* class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for *`Message`*) policy *`Compat32`*. If you are going to use another policy, you should be using the *`EmailMessage`* class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5322** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *`multipart/*`* or *`message/rfc822`*.

The conceptual model provided by a *`Message`* object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a

serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The *Message* pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the *Message* class:

class email.message.**Message**(*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

在 3.3 版的變更: 新增 *policy* 關鍵字引數。

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode "unknown character" code points. (See also *as_bytes()* and *BytesGenerator*.)

在 3.4 版的變更: 新增 *policy* 關鍵字引數。

__str__()

Equivalent to *as_string()*. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required

by the Unix mbox format. For more flexibility, instantiate a *BytesGenerator* instance and use its *flatten()* method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

在 3.4 版被加入.

__bytes__()

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

在 3.4 版被加入.

is_multipart()

Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *Message* is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)

Add the given *payload* to the current payload, which must be `None` or a list of *Message* objects before the call. After the call, the payload will always be a list of *Message* objects. If you want to set the payload to a scalar object (e.g. a string), use *set_payload()* instead.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by *set_content()* and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

Return the current payload, which will be a list of *Message* objects when *is_multipart()* is `True`, or a string when *is_multipart()* is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *get_payload()* will return the *i*-th element of the payload, counting from zero, if *is_multipart()* is `True`. An *IndexError* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *is_multipart()* is `False`) and *i* is given, a *TypeError* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is `True`, then `None` is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (*InvalidBase64PaddingDefect* or *InvalidBase64CharactersDefect*, respectively).

When *decode* is `False` (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the `charset` specified by the *Content-Type* header, using the `replace` error handler. If no `charset` is specified, or if the `charset` given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by *get_content()* and *iter_parts()*.

set_payload(*payload*, *charset=None*)

Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by *set_content()*.

set_charset(*charset*)

Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or `None`. If it is a string, it will be converted to a *Charset* instance. If *charset* is `None`, the `charset` parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its `charset` parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *charset* parameter of the *email.message.EmailMessage.set_content()* method.

get_charset()

Return the *Charset* instance associated with the message's payload.

This is a legacy method. On the *EmailMessage* class it always returns `None`.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of `unknown-8bit`.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant named headers.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()

Return a list of all the message's header field names.

values()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

以下是個範例：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a *KeyError* is raised.

get_content_type()

Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by *get_default_type()* will be returned. Since according to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

get_content_maintype()

Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()

Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

Return the message's *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in *get_param()* and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling *email.utils.collapse_rfc2231_value()*, passing in the return value from *get_param()*. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header as not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to *Content-Type*, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

在 3.4 版的變更: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to *Content-Type*.

set_type(*type*, *header='Content-Type'*, *requote=True*)

Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a *ValueError* is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)

Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)

Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)

Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of

the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset(*failobj=None*)

Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

Note that this method differs from *get_charset()* which returns the *Charset* instance for the default encoding of the message body.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

在 3.5 版被加入.

walk()

The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the *Generator* to print a newline at the end of the file.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

20.1.10 email.mime：從頭開始建立電子郵件和 MIME 物件

原始碼： Lib/email/mime/

This module is part of the legacy (`Compat32`) email API. Its functionality is partially replaced by the *contentmanager* in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual *Message* objects by hand. In fact, you can also take an existing structure and add new *Message* objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating *Message* instances, adding attachments and all the appropriate headers manually. For MIME messages though, the *email* package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)

模組：*email.mime.base*

This is the base class for all the MIME-specific subclasses of *Message*. Ordinarily you won't create instances specifically of *MIMEBase*, although you could. *MIMEBase* is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the *Content-Type* major type (e.g. *text* or *image*), and *_subtype* is the *Content-Type* minor type (e.g. *plain* or *gif*). *_params* is a parameter key/value dictionary and is passed directly to *Message.add_header*.

If *policy* is specified, (defaults to the *compat32* policy) it will be passed to *Message*.

The *MIMEBase* class always adds a *Content-Type* header (based on *_maintype*, *_subtype*, and *_params*), and a *MIME-Version* header (always set to `1.0`).

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.nonmultipart.**MIMENonMultipart**

模組: *email.mime.nonmultipart*

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the *attach()* method, which only makes sense for *multipart* messages. If *attach()* is called, a *MultipartConversionError* exception is raised.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)

模組: *email.mime.multipart*

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When `None` (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the *Message.attach* method.

Optional *policy* argument defaults to *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模組: *email.mime.application*

A subclass of *MIMENonMultipart*, the *MIMEApplication* class is used to represent MIME message objects of major type *application*. *_data* contains the bytes for the raw application data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the *MIMEApplication* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模組: *email.mime.audio*

A subclass of *MIMENonMultipart*, the *MIMEAudio* class is used to create MIME message objects of major type *audio*. *_audiodata* contains the bytes for the raw audio data. If this data can be decoded as au, wav, aiff, or aifc, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the audio subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the *MIMEAudio* instance. It should use

get_payload() and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the base class constructor.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模組: *email.mime.image*

A subclass of *MIMENonMultipart*, the *MIMEImage* class is used to create MIME message objects of major type *image*. *_imagedata* contains the bytes for the raw image data. If this data type can be detected (jpeg, png, gif, tiff, rgb, pbm, pgm, ppm, rast, xbm, bmp, webp, and exr attempted), then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the image subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the *MIMEImage* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

Optional *policy* argument defaults to *compat32*.

_params are passed straight through to the *MIMEBase* constructor.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

模組: *email.mime.message*

A subclass of *MIMENonMultipart*, the *MIMEMessage* class is used to create MIME objects of main type *message*. *_msg* is used as the payload, and must be an instance of class *Message* (or a subclass thereof), otherwise a *TypeError* is raised.

Optional *_subtype* sets the subtype of the message; it defaults to *rfc822*.

Optional *policy* argument defaults to *compat32*.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

模組: *email.mime.text*

A subclass of *MIMENonMultipart*, the *MIMEText* class is used to create MIME objects of major type *text*. *_text* is the string for the payload. *_subtype* is the minor type and defaults to *plain*. *_charset* is the character set of the text and is passed as an argument to the *MIMENonMultipart* constructor; it defaults to `us-ascii` if the string contains only `ascii` code points, and `utf-8` otherwise. The *_charset* parameter accepts either a string or a *Charset* instance.

Unless the *_charset* argument is explicitly set to `None`, the MIMEText object created will have both a *Content-Type* header with a `charset` parameter, and a *Content-Transfer-Encoding* header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a charset is passed in the `set_payload` command. You can "reset" this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new *Content-Transfer-Encoding* header).

Optional *policy* argument defaults to *compat32*.

在 3.5 版的變更: *_charset* also accepts *Charset* instances.

在 3.6 版的變更: 新增僅限關鍵字參數 *policy*。

20.1.11 email.header：國際化標頭

原始碼： Lib/email/header.py

This module is part of the legacy (`Compat32`) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the *`EmailMessage`* class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The *`email`* package supports these standards in its *`email.header`* and *`email.charset`* modules.

If you want to include non-ASCII characters in your email headers, say in the *`Subject`* or *`To`* fields, you should use the *`Header`* class and assign the field in the *`Message`* object to an instance of *`Header`* instead of using a string for the header value. Import the *`Header`* class from the *`email.header`* module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Notice here how we wanted the *`Subject`* field to contain a non-ASCII character? We did this by creating a *`Header`* instance and passing in the character set that the byte string was encoded in. When the subsequent *`Message`* instance was flattened, the *`Subject`* field was properly **RFC 2047** encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the *`Header`* class description:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with *`append()`* method calls. *s* may be an instance of *`bytes`* or *`str`*, but see the *`append()`* documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the *`append()`* method. It also sets the default character set for all subsequent *`append()`* calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the `us-ascii` character set is used both as *s*'s initial charset and as the default for subsequent *`append()`* calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. *`Subject`*) pass in the name of the field in *header_name*. The default *maxlinelen* is 78, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be **RFC 2822**-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the `append()` method.

append(*s*, *charset=None*, *errors='strict'*)

Append the string *s* to the MIME header.

Optional *charset*, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be an instance of `bytes` or `str`. If it is an instance of `bytes`, then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is an instance of `str`, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an **RFC 2822**-compliant header using **RFC 2047** rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822**'s 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

在 3.2 版的變更: 新增引數 *linesep*。

The `Header` class also provides a number of methods to support standard operators and built-in functions.

__str__()

Returns an approximation of the `Header` as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

在 3.2 版的變更: Added handling for the `'unknown-8bit'` charset.

__eq__(*other*)

This method allows you to compare two `Header` instances for equality.

__ne__(*other*)

This method allows you to compare two `Header` instances for inequality.

The `email.header` module also provides the following convenient functions.

email.header.**decode_header**(*header*)

Decode a message header value without converting the character set. The header value is in *header*.

For historical reasons, this function may return either:

1. A list of pairs containing each of the decoded parts of the header, `(decoded_bytes, charset)`, where *decoded_bytes* is always an instance of `bytes`, and *charset* is either:
   - A lower case string containing the name of the character set specified.
   - `None` for non-encoded parts of the header.

2. A list of length 1 containing a pair `(string, None)`, where *string* is always an instance of *str*.

An *email.errors.HeaderParseError* may be raised when certain decoding errors occur (e.g. a base64 decoding exception).

以下🄵範例：

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
>>> decode_header('unencoded_string')
[('unencoded_string', None)]
>>> decode_header('bar =?utf-8?B?ZsOzbw==?=')
[(b'bar ', None), (b'f\xc3\xb3o', 'utf-8')]
```

備🄵

This function exists for backwards compatibility only. For new code, we recommend using *email.headerregistry.HeaderRegistry*.

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

備🄵

This function exists for backwards compatibility only, and is not recommended for use in new code.

20.1.12 email.charset：字元集合的表示

原始碼： Lib/email/charset.py

此模組是舊版 (`Compat32`) email API 的其中一部份，在新版的 API 只有使用🄵名表。

此章節的其余🄵容是模組的原始🄵明文件

此模組提供一個類🄵*Charset* 來表示電子郵件訊息中的字元集合和字元集合轉🄵，也包含字元集合登🄵檔 (registry) 和其他許多運用此登🄵檔的便捷方法。*Charset* 的實例被*email* 套件中其他數個模組所使用。

從*email.charset* 模組中引入此類🄵

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

將字元集合對映到其電子郵件特性 (properties)。

此類🄵提供有關於電子郵件對特定字元集合的規範資訊。考慮到適用之編解碼器 (codec) 的可用性，它還提供了在字元集合之間進行轉🄵的便利例行操作 (routine)。給定一個字元集合，它將盡量提供有關如何以符合 RFC (RFC-compliant) 的方式在電子郵件訊息中使用該字元集合的資訊。

在電子郵件標頭 (header) 或🄵文 (body) 用特定字元集合時必須使用可列印字元編碼 (quoted-printable) 或 base64 來編碼。特定字元集合不允許出現在電子郵件中、必須被徹底轉🄵。

可選的 *input_charset* 描述如下；*input_charset* 會被🄵制轉🄵 (coerced) 🄵小寫。經過🄵名標准化 (alias normalize) 後，會進到字元集合登🄵檔 (registry) 去查詢此字元集合使用的標頭編碼、🄵文編碼以及輸出轉🄵編解碼器。舉例來🄵，如果 *input_charset* 是 `iso-8859-1`，標頭跟🄵文會以可列印字元編

碼🄵且不需要輸出轉🄵編解碼器。如果 *input_charset* 是 `euc-jp`，標頭則會被編碼成 base64、🄵文不會被編碼，但輸出文字則會從 `euc-jp` 字元集合被轉🄵成 `iso-2022-jp` 字元集合。

Charset 實例有以下資料屬性：

input_charset

指定的初始字元集合。通用🄵名會被轉🄵成它們的官方電子郵件名稱（例如：`latin_1` 會被轉🄵成 `iso-8859-1`）。預設🄵 7 位元 `us-ascii`。

header_encoding

如果字元集合在被用於電子郵件標頭前必須被編碼的話，這個屬性會被設🄵 `charset.QP`（表示可列印字元編碼）、`charset.BASE64`（表示 base64 編碼格式）或是 `charset.SHORTEST` 表示最短的 QP 或是 base64 編碼格式。不然這個屬性會是 `None`。

body_encoding

與 *header_encoding* 相同，但表示郵件訊息🄵文的編碼，與上述的標頭編碼有可能不同。`charset.SHORTEST` 是不允許於 *body_encoding* 使用的。

output_charset

部分的字元集合在用於電子郵件的標頭或🄴文前必須先被轉🄵。如果 *input_charset* 是這些字元集合的其中之一，這個屬性將會包含輸出時轉🄴成的字元集合名稱。不然這個屬性則🄵 `None`。

input_codec

用於將 *input_charset* 轉🄵成 Unicode 的 Python 編解碼器的名稱。如果不需要轉🄵編解碼器，這個屬性🄵 `None`。

output_codec

用於將 Unicode 轉🄵成 *output_charset* 的 Python 編解碼器名稱。如果不需要轉🄵編解碼器，這個屬性將會與 *input_codec* 有相同的值。

Charset 實例還有以下方法：

get_body_encoding()

回傳用於🄵文編碼的🄵容傳輸編碼 (content transfer encoding)。

這可以是字串 `quoted-printable` 或 `base64`，具體取🄵於所使用的編碼，或者它也可以是一個函式，在這種情🄵下，你應該使用單個引數呼叫該函式，即正被編碼的 Message 物件。然後函式應將 *Content-Transfer-Encoding* 標頭本身設定🄵任何適當的值。

如果 *body_encoding* 🄵 `QP` 則回傳字串 `quoted-printable`，如果 *body_encoding* 🄵 `BASE64` 則回傳字串 `base64`，否則回傳字串 `7bit` 。

get_output_charset()

回傳輸出字元集合。

如果不🄵 `None` 則這會是 *output_charset* 屬性，否則它是 *input_charset*。

header_encode(*string*)

對字串 *string* 進行標頭編碼 (header-encode)。

編碼類型（base64 或可列印字元編碼）將基於 *header_encoding* 屬性。

header_encode_lines(*string*, *maxlengths*)

透過先將 *string* 轉🄵🄵位元組來對它進行標頭編碼。

這與*header_encode()* 類似，只不過字串不會超過由引數 *maxlengths* 給定的最大列長度 (maximum line length)，該引數必須是個🄴代器：從此🄴代器回傳的每個元素將提供下一個最大列長度。

body_encode(*string*)

對字串 *string* 進行🄵文編碼 (body-encode)。

編碼類型（base64 或可列印字元編碼）將基於 *body_encoding* 屬性。

Charset 類🄵也提供了許多支援標准操作和🄵建函式的方法。

__str__()
回傳🄵制轉🄵 *input_charset* 🄵小寫後的字串。__repr__() 是 __str__() 的🄵名。

__eq__(*other*)
此方法讓你比較兩個*Charset* 實例的相等性。

__ne__(*other*)
此方法讓你比較兩個*Charset* 實例的不相等性。

email.charset 模組還提供以下函式，用於將項目新增至全域字元集合、🄵名和編解碼器登🄵檔中：

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
將字元特性 (properties) 新增至全域登🄵檔。

charset 是輸入的字元集合，且必須是字元集合的規範名稱 (canonical name)。

可選的 *header_enc* 和 *body_enc* 🄵 charset.QP（表示可列印字元編碼）、charset.BASE64（表示 base64 編碼）、charset.SHORTEST（表示最短的 base64 或可列印字元編碼）或 None（表示無編碼）。SHORTEST 僅在用於 *header_enc* 時有效。預設🄵 None，表示無編碼。

可選的 *output_charset* 是輸出應🄵用的字元集合。當呼叫 Charset.convert() 方法時，將從輸入字元集合轉🄵🄵 Unicode，再轉🄵🄵輸出字元集合。預設是以與輸入相同的字元集合輸出。

input_charset 和 *output_charset* 都必須在模組的字元集合到編解碼器對映 (character set-to-codec mapping) 中具有 Unicode 編解碼器項目；使用*add_codec()* 來新增模組未知的編解碼器。有關更多資訊請參🄵*codecs* 模組的文件。

全域字元集合登🄵檔保存在模組全域字典 CHARSETS 中。

email.charset.**add_alias**(*alias*, *canonical*)
新增字元集合🄵名。*alias* 是🄵名，例如 latin-1。*canonical* 是字元集合的規範名稱，例如 iso-8859-1。

全域字元集合🄵名登🄵檔保存在模組全域字典 ALIASES 中。

email.charset.**add_codec**(*charset*, *codecname*)
新增一個編解碼器，將給定字元集合中的字元與 Unicode 進行對映。

charset 是字元集合的規範名稱。*codecname* 是 Python 編解碼器的名稱，適用於*str* 的*encode()* 方法的第二個引數。

20.1.13 email.encoders：編碼器

原始碼： Lib/email/encoders.py

此模組是舊版 (Compat32) 電子郵件 API 的一部分。在新 API 中，此功能由*set_content()* 方法的 *cte* 參數提供。

此模組在 Python 3 中已🄵用。不應明確呼叫此處提供的函式，因🄵*MIMEText* 類🄴使用在該類🄵的實例化過程中傳遞的 *_subtype* 和 *_charset* 值來設定🄵容類型 (content type) 和 CTE 標頭。

本節中的其余文字是該模組的原始文件。

從零開始建立*Message* 物件時，你通常需要對負載 (payload) 進行編碼，以便透過相容的郵件伺服器進行傳輸。對於包含二進位資料的 *image/** 和 *text/** 類型的訊息尤其如此。

email 套件在其*encoders* 模組中提供了一些方便的編碼器。這些編碼器實際上由*MIMEAudio* 和*MIMEImage* 類🄵建構函式使用來提供預設編碼。所有編碼器函式都只接受一個引數，也就是要編碼的訊息物件。他們通常會提取負載、對其進行編碼，然後將負載重設🄵新編碼的值。他們也應適當地設定 *Content-Transfer-Encoding* 標頭。

請注意，這些函式對於多部分訊息 (multipart message) 🄵有意義。它們必須應用於各個子部分，如果傳遞類型🄵多部分訊息，則會引發*TypeError*。

以下是提供的編碼函式：

email.encoders.**encode_quopri**(*msg*)

將負載編碼🄵可列印字元 (quoted-printable) 形式，🄵將 *Content-Transfer-Encoding* 標頭設定🄵 `quoted-printable`[1]。當大部分負載是正常的可列印資料，但包含一些不可列印的字元時，這是一種很好的編碼。

email.encoders.**encode_base64**(*msg*)

將負載編碼🄵 base64 形式，🄵將 *Content-Transfer-Encoding* 標頭設定🄵 `base64`。當大部分負載是不可列印資料時，這是一種很好的編碼，因🄵它是比可列印字元更緊🄵的形式。Base64 編碼的缺點是它使文字無法讓人類可讀。

email.encoders.**encode_7or8bit**(*msg*)

這實際上🄵🄵有修改訊息的負載，但它確實根據負載資料將 *Content-Transfer-Encoding* 標頭設定🄵適當的 `7bit` 或 `8bit`。

email.encoders.**encode_noop**(*msg*)

這🄵有任何作用；它甚至🄵有設定 *Content-Transfer-Encoding* 標頭。

20.1.14 email.utils：雜項工具

原始碼： Lib/email/utils.py

There are a couple of useful utilities provided in the *email.utils* module:

email.utils.**localtime**(*dt=None*)

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a *datetime* instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, `dt.tzinfo` is `None`), it is assumed to be in local time.

在 3.3 版被加入.

Deprecated since version 3.12, removed in version 3.14: The *isdst* parameter.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)

Returns a string suitable for an **RFC 2822**-compliant *Message-ID* header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the '@'. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts.

在 3.2 版的變更: 新增 *domain* 關鍵字。

The remaining functions are part of the legacy (`Compat32`) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

email.utils.**quote**(*str*)

Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

email.utils.**unquote**(*str*)

Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

email.utils.**parseaddr**(*address*, ***, *strict=True*)

Parse address -- which should be the value of some address-containing field such as *To* or *Cc* -- into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

If *strict* is true, use a strict parser which rejects malformed inputs.

在 3.13 版的變更: Add *strict* optional parameter and reject malformed inputs by default.

[1] 請注意，使用*encode_quopri()* 進行編碼也會對資料中的所有定位字元 (tab) 和空格字元進行編碼。

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

The inverse of *parseaddr()*, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a `To` or `Cc` header. If the first element of *pair* is false, then the second element is returned unmodified.

Optional *charset* is the character set that will be used in the **RFC 2047** encoding of the `realname` if the `realname` contains non-ASCII characters. Can be an instance of *str* or a *Charset*. Defaults to `utf-8`.

在 3.3 版的變更: 新增 *charset* 選項。

email.utils.**getaddresses**(*fieldvalues*, *, *strict=True*)

This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by *Message.get_all*.

If *strict* is true, use a strict parser which rejects malformed inputs.

Here's a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

在 3.13 版的變更: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**parsedate**(*date*)

Attempts to parse a date according to the rules in **RFC 2822**. however, some mailers don't follow that format as specified, so *parsedate()* tries to guess correctly in such cases. *date* is a string containing an **RFC 2822** date, such as `"Mon, 20 Nov 1995 19:12:08 -0500"`. If it succeeds in parsing the date, *parsedate()* returns a 9-tuple that can be passed directly to *time.mktime()*; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_tz**(*date*)

Performs the same function as *parsedate()*, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is `0`, which represents UTC. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_to_datetime**(*date*)

The inverse of *format_datetime()*. Performs the same function as *parsedate()*, but on success returns a *datetime*; otherwise `ValueError` is raised if *date* contains an invalid value such as an hour greater than 23 or a timezone offset not between -24 and 24 hours. If the input date has a timezone of `-0000`, the `datetime` will be a naive `datetime`, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the `datetime` will be an aware `datetime` with the corresponding a *timezone tzinfo*.

在 3.3 版被加入.

email.utils.**mktime_tz**(*tuple*)

Turn a 10-tuple as returned by *parsedate_tz()* into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

Returns a date string as per **RFC 2822**, e.g.:

[1] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822**.

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating-point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional *localtime* is a flag that when `True`, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional *usegmt* is a flag that when `True`, outputs a date string with the timezone as an ascii string `GMT`, rather than a numeric `-0000`. This is needed for some protocols (such as HTTP). This only applies when *localtime* is `False`. The default is `False`.

`email.utils.`**`format_datetime`**(*dt*, *usegmt=False*)

Like `formatdate`, but the input is a `datetime` instance. If it is a naive datetime, it is assumed to be "UTC with no information about the source timezone", and the conventional `-0000` is used for the timezone. If it is an aware `datetime`, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then *usegmt* may be set to `True`, in which case the string `GMT` is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

在 3.3 版被加入.

`email.utils.`**`decode_rfc2231`**(*s*)

Decode the string *s* according to **RFC 2231**.

`email.utils.`**`encode_rfc2231`**(*s*, *charset=None*, *language=None*)

Encode the string *s* according to **RFC 2231**. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

`email.utils.`**`collapse_rfc2231_value`**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

When a header parameter is encoded in **RFC 2231** format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional *errors* is passed to the *errors* argument of `str`'s `encode()` method; it defaults to `'replace'`. Optional *fallback_charset* specifies the character set to use if the one in the **RFC 2231** header is not known by Python; it defaults to `'us-ascii'`.

For convenience, if the *value* passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

`email.utils.`**`decode_params`**(*params*)

Decode parameters list according to **RFC 2231**. *params* is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

20.1.15 `email.iterators`: 代器

原始碼： Lib/email/iterators.py

Iterating over a message object tree is fairly easy with the `Message.walk` method. The `email.iterators` module provides some useful higher level iterations over message object trees.

`email.iterators.`**`body_line_iterator`**(*msg*, *decode=False*)

This iterates over all the payloads in all the subparts of *msg*, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn't a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional *decode* is passed through to `Message.get_payload`.

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

This iterates over all the subparts of *msg*, returning only those subparts that match the MIME type specified by *maintype* and *subtype*.

Note that *subtype* is optional; if omitted, then subpart MIME type matching is done only with the main type. *maintype* is optional too; it defaults to `text`.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of `text/*`.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python's `print()` function. *level* is used internally. *include_default*, if true, prints the default type as well.

也參考

***smtplib* 模組**
SMTP (Simple Mail Transport Protocol) client

***poplib* 模組**
POP (Post Office Protocol) client

***imaplib* 模組**
IMAP (Internet Message Access Protocol) client

***mailbox* 模組**
Tools for creating, reading, and managing collections of messages on disk using a variety standard formats.

20.2 `json` --- JSON 編碼器與解碼器

原始碼： Lib/json/__init__.py

JSON (JavaScript Object Notation) 是一個輕量化的資料交🄵格式，在 **RFC 7159**（其廢🄵了 **RFC 4627**）及 ECMA-404 🄵面有詳細🄵明，它🄵發自 JavaScript 的物件字面語法 (object literal syntax)（雖然它🄵不是 JavaScript 的嚴格子集[1]）。

[1] 如 RFC 7159 更正 所述，JSON 允許字串中出現 U+2028（列分隔符）和 U+2029（段落分隔符）字元，而 JavaScript（截至 ECMAScript 5.1 版）則不允許。

備🄵

The term ”object” in the context of JSON processing in Python can be ambiguous. All values in Python are objects. In JSON, an object refers to any data wrapped in curly braces, similar to a Python dictionary.

警告

當剖析無法信任來源的 JSON 資料時要小心。一段惡意的 JSON 字串可能會導致解碼器耗費大量 CPU 與記憶體資源。建議限制剖析資料的大小。

此模組🄵習慣標准函式庫*marshal* 與*pickle* 模組的使用者提供熟悉的 API。

對基本 Python 物件階層進行編碼：

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

改用緊🄵型編碼方式：

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

美化輸出：

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

自訂 JSON 物件編碼方式：

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

JSON 解碼：

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

自訂 JSON 物件解碼方式：

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

繼承*JSONEncoder*類🄵🄵自行擴充額外的編碼方法：

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return super().default(obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

在 shell 使用*json*來驗證和美化印出：

```
$ echo '{"json":"obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

更詳盡的文件請見命令列介面。

備🄵

JSON 語法是 YAML 1.2 語法的一種子集合。所以如果使用預設的設定的話（准確來🄴，使用預設的 *separators* 分隔符設定的話），這個模組的輸出也符合 YAML 1.0 和 1.1 的子集合規範。因此你也可以利用這個模組來當作 YAML 的序列化工具（serializer）。

備🄵

這個模組的編、解碼器預設會保存輸入與輸出資料的順序關🄵，除非一開始的輸入本身就是無序的。

20.2.1 基本用法

json.**dump**(*obj*, *fp*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

參考這個*Python-to-JSON 轉🄵表*將 *obj* 序列化🄵符合 JSON 格式的串流，🄵寫入到 *fp*（一個支援 `.write()` 方法的*file-like object*）

備🄵

與*pickle* 和*marshal* 不同，JSON 不具有二進位分框（binary framed）的協定，因此嘗試重🄵呼叫*dump()* 來序列化多個物件到同一個 *fp* 🄵將導致無效的 JSON 檔案。

參數

- **obj** (object) -- 要被序列化的 Python 物件。
- **fp** (*file-like object*) -- The file-like object *obj* will be serialized to. The `json` module always produces *str* objects, not *bytes* objects, therefore `fp.write()` must support *str* input.
- **skipkeys** (bool) -- If `True`, keys that are not of a basic type (*str*, *int*, *float*, *bool*, `None`) will be skipped instead of raising a *TypeError*. Default `False`.
- **ensure_ascii** (bool) -- If `True` (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `False`, these characters will be outputted as-is.
- **check_circular** (bool) -- 如🄵 `False`，則針對不同容器型🄵的循環參照 (circular reference) 檢查將會被跳過，若有循環參照則最後將引發*RecursionError*（或其他更糟的錯誤）。預設🄵 `True`。
- **allow_nan** (bool) -- 如🄵 `False`，則序列化不符合嚴格 JSON 規範的超出範圍*float* 值 (`nan`, `inf`, `-inf`) 會引發*ValueError*。如🄵 `True`（預設值），則將使用它們的 JavaScript 等效表示 (`NaN`, `Infinity`, `-Infinity`)。
- **cls** (a *JSONEncoder* subclass) -- If set, a custom JSON encoder with the *default()* method overridden, for serializing into custom datatypes. If `None` (the default), `JSONEncoder` is used.
- **indent** (*int* | *str* | *None*) -- If a positive integer or string, JSON array elements and object members will be pretty-printed with that indent level. A positive integer indents that many spaces per level; a string (such as `"\t"`) is used to indent each level. If zero, negative, or `""` (the empty string), only newlines are inserted. If `None` (the default), the most compact representation is used.
- **separators** (*tuple* | *None*) -- 一個二元組：`(item_separator, key_separator)`。如🄵 `None`（預設值）則 *separators* 被預設🄵 `(', ', ': ')`，否則預設🄵 `(',', ': ')`。想要獲得最緊🄵的 JSON 表示形式，你可以指定 `(',', ':')` 來消除空格。
- **default** (*callable* | None) -- A function that is called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a *TypeError*. If `None` (the default), `TypeError` is raised.
- **sort_keys** (bool) -- If `True`, dictionaries will be outputted sorted by key. Default `False`.

在 3.2 版的變更: 除了整數之外，*indent* 還允許使用字串作🄵輸入。

在 3.4 版的變更: 如果 *indent* 不是 `None`，則使用 `(',', ': ')` 作🄵預設值

在 3.6 版的變更: 所有可選參數現在都是僅限關鍵字參數了。

`json.`**`dumps`**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

使用此轉🄵表來將 *obj* 序列化🄵 JSON 格式 *str*。這個引數的作用與 *dump()* 中的同名引數意義相同。

> **備🄵**
>
> JSON 鍵/值對中的鍵始終🄵 *str* 型🄵。當字典被轉🄵🄵 JSON 時，字典的所有鍵值資料型🄵都會被🄵制轉🄵🄵字串。因此，如果將字典先轉🄵🄵 JSON 格式然後再轉🄵回字典，則該字典可能不等於原始字典。也就是🄵，如果字典 x 含有非字串鍵值，則 `loads(dumps(x)) != x`。

`json.`**`load`**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

使用此*JSON-to-Python 轉🄵表*將 *fp* 解碼🄵 Python 物件。

參數

- **`fp`** (*file-like object*) -- A `.read()`-supporting *text file* or *binary file* containing the JSON document to be deserialized.
- **`cls`** (a *JSONDecoder* subclass) -- If set, a custom JSON decoder. Additional keyword arguments to `load()` will be passed to the constructor of *cls*. If `None` (the default), `JSONDecoder` is used.
- **`object_hook`** (*callable* | None) -- 如果有設定，會用解碼後的 JSON 物件字面值的結果（一個 *dict*）來呼叫此函式。此函式的回傳值將用來取代 *dict*。此功能可用於實作自訂解碼器，例如 JSON-RPC 類🄵提示。預設🄵 `None`。
- **`object_pairs_hook`** (*callable* | None) -- 如果有設定，會用任何 JSON 物件字面值以有序對串列解碼的結果來呼叫此函式。此函式的回傳值將用來取代 *dict*。此功能可用於實作自訂解碼器。如果也設定了 *object_hook*，以 *object_pairs_hook* 優先。預設🄵 `None`。
- **`parse_float`** (*callable* | None) -- 如果有設定，會用每個要被解碼的 JSON 浮點數字串來呼叫此函式。如果🄵 `None`（預設值），則等同於 `float(num_str)`。此功能可用於將 JSON 浮點數剖析🄵自訂資料型🄵，例如 *decimal.Decimal*。
- **`parse_int`** (*callable* | None) -- 如果有設定，會用每個要被解碼的 JSON 整數字串來呼叫此函式。如果🄵 `None`（預設值），則等同於 `int(num_str)`。此功能可用於將 JSON 整數剖析🄵自訂資料型🄵，例如 *float*。
- **`parse_constant`** (*callable* | None) -- 如果有設定，會用以下字串之一來呼叫此函式：`'-Infinity'`、`'Infinity'` 或 `'NaN'`。此功能可用於在遇到無效的 JSON 數字時引發例外。預設🄵 `None`。

引發

- ***JSONDecodeError*** -- 當被去序列化的資料不是有效的 JSON 文件時。
- ***UnicodeDecodeError*** -- 當被去序列化的資料不包含 UTF-8、UTF-16 或 UTF-32 編碼資料時。

在 3.1 版的變更:

- 新增可選的 *object_pairs_hook* 參數。
- 遭遇'null'、'true' 或'false' 時不再以 *parse_constant* 給定的函式來處理了。

在 3.6 版的變更:

- 所有可選參數現在都是僅限關鍵字參數了。

- 現在，*fp* 可以是一個二進位檔案，前提是其編碼格式🄵 UTF-8、UTF-16 或 UTF-32。

在 3.11 版的變更: 預設 *parse_int* 使用的`int()`函式現在有限制整數字串的長度上限了，限制由直譯器的整數字串轉🄵長度限制機制來達成，這能防止阻斷服務攻擊 (Denial of Service attacks)。

`json.`**`loads`**(*s*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

與`load()`相同，但不是類檔案物件，而是使用轉🄵表將 *s*（一個包含 JSON 文件的`str`、`bytes`或`bytearray`實例）去序列化🄵 Python 物件。

在 3.6 版的變更: 現在，*s* 可以是一個二進位檔案如`bytes`或`bytearray`，前提是其編碼格式🄵 UTF-8、UTF-16 或 UTF-32。

在 3.9 版的變更: 🄵除關鍵字引數 *encoding*。

20.2.2 編碼器與解碼器

`class` `json.`**`JSONDecoder`**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

簡易 JSON 解碼器

預設將執行下列資料型🄵轉🄵:

JSON	Python
object	dict
array	list
string	str
number (整數)	int
number (實數)	float
true	True
false	False
null	None

雖然 `NaN`、`Infinity` 和 `-Infinity` 🄵不符合 JSON 規範，但解碼器依然能正確地將其轉🄵到相應的 Python `float` 值。

object_hook 是一個可選函式，其接受一個解碼後的 JSON 物件作🄵輸入，🄵使用其回傳值來取代原先的`dict`。這個功能可用於提供自訂的去序列化（例如支援 JSON-RPC 類🄵提示）。

object_pairs_hook is an optional function that will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

在 3.1 版的變更: 新增對於 *object_pairs_hook* 的支援。

parse_float 🄵可選函式，每個要被解碼的 JSON 浮點數字串都會改用這個參數給定的函式來進行解碼。預設情🄵這等效於 `float(num_str)`。這個參數可用於將 JSON 中的浮點數解碼或剖析🄵另一種資料型🄵（例如`decimal.Decimal`）。

parse_int 🄵可選函式，當解碼 JSON 整數字串時會被呼叫。預設情🄵等效於 `int(num_str)`。這個參數可用於將 JSON 中的整數解碼或剖析🄵另一種資料型🄵（例如`float`）。

parse_constant 🄵可選函式，在解碼時若遭遇字串 `'-Infinity'`、`'Infinity'` 或 `'NaN'` 其中之一則會改用這個參數給定的函式來進行解碼。這也可用於使解碼過程中遇到無效的 JSON 數字時引發一個例外。

如果 *strict* 被設🄵 false（預設值🄵 `True`），那🄵字串中將允許控制字元。此語境中的控制字元指的是 ASCII 字元編碼在 0 ~ 31 範圍🄵的字元，包括 `'\t'`（tab）、'n'、'r' 和 `'\0'`。

如果被去序列化（deserialized）的資料不符合 JSON 格式，將會引發`JSONDecodeError`例外。

在 3.6 版的變更: 所有參數現在都是僅限關鍵字參數了。

decode(*s*)

回傳用 Python 型式表達的 *s*（一個含有 JSON 文件的*str* 實例）。

若給定的輸入不符合 JSON 格式會引發*JSONDecodeError* 例外。

raw_decode(*s*)

將 *s*（一個開頭部分含有合格 JSON 文件的*str*）解碼，🄵將 JSON 文件結束點的索引值（index）和解碼結果合🄵🄵一個二元組（2-tuple）後回傳。

這個方法可以用來解碼尾段可能帶有 JSON 以外資料的文字。

class json.**JSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

可擴充的 Python 資料結構 JSON 編碼器。

預設可支援下列物件及型🄵:

Python	JSON
dict	object
list, tuple	array
str	string
int、float 或可作🄵整數或浮點數運算的衍生列舉（int- or float-derived Enums）	number
True	true
False	false
None	null

在 3.4 版的變更: 增加對整數（int）、浮點數（float）或可作🄵整數或浮點數運算的衍生列舉（int- or float-derived Enums）類🄵的支援性。

若要擴充此功能來識🄵其他物件，請繼承🄵實作一個*default()* 方法。此方法應回傳一個可序列化的 o 物件，否則此方法應呼叫父類🄵的 JSONEncoder.default 方法（以引發*TypeError* 例外）。

若 *skipkeys* 🄵 false（預設值），則當在編碼不是*str*、*int*、*float*、*bool* 或 `None` 的鍵值時，將引發*TypeError*。如果 *skipkeys* 🄵 true，這些項目將直接被跳過。

如果 *ensure_ascii* 被設🄵 true（預設值），則輸出時將確保所有輸入的非 ASCII 字元都會被跳🄵。若 *ensure_ascii* 🄵 false，則這些字元將照原樣輸出。

如果 *check_circular* 🄵 true（預設值），則會在編碼期間檢查串列（list）、字典（dict）和自訂編碼物件的循環參照，以防止無限遞🄵（一個會導致*RecursionError* 例外的問題）。否則不會進行此類檢查。

如果 *allow_nan* 🄵 true（預設值），則 `NaN`、`Infinity` 和 `-Infinity` 將按照原樣進行編碼。請記得此行🄵不符合標准 JSON 規範，但的確與大多數基於 JavaScript 的編碼器和解碼器一致。否則若設🄵 false，嘗試對這些浮點數進行編碼將引發*ValueError* 例外。

如果 *sort_keys* 🄵 true（預設值: `False`），則 dictionary（字典）的輸出將按鍵值排序。這項功能可確保 JSON 序列化的結果能被互相比較，能讓日常的回歸測試檢查變得方便一些。

如果 *indent* 是非負整數或字串，則 JSON 陣列元素和物件成員將使用該縮排等級進行格式美化。縮排等級 0、負數或 `""` 只會插入🄵行符號。`None`（預設值）等於是選擇最緊🄵的表示法。使用正整數縮排可以在每層縮排數量相同的空格。如果 *indent* 是一個字串（例如 `"\t"`），則該字串用於縮排每個層級。

在 3.2 版的變更: 除了整數之外，*indent* 還允許使用字串作🄵輸入。

如果有指定本引數🄵容，*separators* 應該是一個 `(item_separator, key_separator)` 二元組。如果 *indent* 🄵 `None` 則預設🄵 `(', ', ': ')`，否則預設🄵 `(',', ': ')`。想要獲得最緊🄵的 JSON 表示形式，你可以改成指定 `(',', ':')` 來消除空格。

在 3.4 版的變更: 如果 *indent* 不是 `None`，則使用 `(',', ': ')` 作🄵預設值

如果有指定本參數，*default* 會是一個遭遇無法序列化的物件時會被呼叫的函式。它應該回傳該物件的 JSON 可編碼版本或引發*TypeError*。如果未指定，則會直接引發*TypeError*。

在 3.6 版的變更: 所有參數現在都是僅限關鍵字參數了。

default(*o*)

在任意一個子類🄵🄵實作這個方法時須讓其回傳一個可序列化的物件 *o*，或呼叫原始的實作以引發 *TypeError* 例外。

舉例來🄵，想要讓編碼器支援任意🄵代器（iterator），你可以實作這樣子的*default()*：

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return super().default(o)
```

encode(*o*)

回傳一個 Python 資料結構物件 *o* 的 JSON 的字串表示。例如：

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

將物件 *o* 編碼，🄵將結果統整🄵一個能依序🄵生（yield）各結果字串的物件。如下例：

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

20.2.3 例外

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)

ValueError 的子類🄵具有下列額外屬性：

msg

未受格式化的錯誤訊息。

doc

正在被剖析的 JSON 文件。

pos

doc 剖析失敗處的起始點的索引值。

lineno

pos 所在的列（line）數。

colno

pos 所在的行（column）數。

在 3.5 版被加入.

20.2.4 合規性與互通性（Interoperability）

JSON 格式是由 **RFC 7159** 和 ECMA-404 規範的。本節詳細🄵明了本模組對 RFC 的遵循程度。簡單起見，*JSONEncoder* 和*JSONDecoder* 子類🄵以及未明確提及的參數將不予討論。

這個模組的部份實作🄵未非常嚴格地遵循 RFC 規範。准確來🄵，下列實際實作符合 JavaScript 語法格式，但🄵不符合 JSON 格式：

- 無限（Infinite）和非數字（NaN）值會被接受。
- 同一個物件🄵可以有重🄵的名稱，但只有最後一個同名物件是有效的。

不過 RFC 准許遵循 RFC 的剖析器接受不合規的文字輸入，所以技術上來🄴若以預設設定運作，本模組的去序列化器（deserializer）是符合 RFC 規範的。

字元編碼格式

RFC 要求 JSON 必須以 UTF-8、UTF-16 或 UTF-32 格式編碼。🄵推薦以 UTF-8 編碼以達成最佳的互通性。

RFC 准許但🄵不🄵制編碼器的 *ensure_ascii=True* 行🄵是預設值，但本模組依然實作了此一選項作🄵預設，因此本模組預設會跳🄴所有非 ASCII 字元。

除了 *ensure_ascii* 選項參數之外，本模組嚴格遵循 Python 物件與*Unicode strings* 之間的轉🄵規範，因此🄵不另外處理字元編碼的問題。

RFC 禁止在文件的開頭加上端序記號（Byte Order Mark），因此本模組的序列化器（serializer）也不會在輸出中加入端序記號。RFC 允許但不🄴制 JSON 去序列化器（deserializer）忽略文件初始的端序記號，因此本模組的去序列化器將在遭遇位於文件開頭的端序記號時引發*ValueError* 例外。

RFC 🄵未明確禁止 JSON 文件包含無法對應有效 Unicode 字元的位元組序列（例如未配對的 UTF-16 代理對（surrogate pairs）），但這個特性的確可能會引起相容性問題。預設情🄴下，當原始輸入的*str* 中存在此類序列時，該模組將接受🄴輸出這些序列的編碼位置（code points）。

正負無限與非數值

RFC 不允許表現無限大或非數值（NaN）。但預設情🄴下，這個模組仍接受🄴輸出 `Infinity`、`-Infinity` 和 `NaN`，如同它們是有效的 JSON 數值字面值：

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

在序列化器中，*allow_nan* 參數可以改變這個行🄵。在去序列化器中，*parse_constant* 參數可以改變這個行🄵。

物件🄵重🄵的名稱

RFC 規範僅表明 JSON 物件中的名字應該是唯一的，但🄵有🄴制要求如何處理重🄴的名字。預設情🄴下，本模組不會因此引發例外；相反的，它會忽略該名字的所有重🄴鍵值對，🄵只保留最後一個：

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

object_parts_hook 參數可以改變這個行🄵。

位於頂層的非物件及非列表值

由已廢🄴的 **RFC 4627** 所規範的舊版 JSON 要求 JSON 文字的頂層值必須是 JSON 物件或陣列（Python *dict* 或*list*），而且不能是 JSON 的 null、boolean、數字或字串值。**RFC 7159** 移除了這個限制，而本模組的序列化器或去串列化器中未曾實施過該限制。

如果想要最大限度地保留互通性，你可能還是會想要自行施加這個限制。

實作限制

某些 JSON 去序列化器的實作可能會造成下列限制：

- JSON 文件長度上限
- JSON 物件或陣列的最大巢狀層數（level of nesting）限制
- 數字的精准度或範圍
- JSON 字串長度上限

本模組除了 Python 資料型態本身或 Python 直譯器本身的限制以外，不會設定任何此類限制。

將資料序列化🄵 JSON 時，要注意可能會使用該 JSON 輸出的應用程式中的相關限制。特🄵要注意的是，JSON 數字常會被去序列化🄵 IEEE 754 雙精度浮點數（double），🄵因而受到其表示範圍和精度限制的影響。這在序列化極大的 Python *int* 數值、或是序列化特殊數字型🄵的實例時（例如*decimal.Decimal*）尤其重要。

20.2.5 命令列介面

原始碼： Lib/json/tool.py

The *json* module can be invoked as a script via `python -m json` to validate and pretty-print JSON objects. The *json.tool* submodule implements this interface.

如果🄵有指定可選引數 `infile` 和 `outfile`，則*sys.stdin* 和*sys.stdout* 將各自做🄵輸入和輸出的預設值。

```
$ echo '{"json": "obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

在 3.5 版的變更: 現在開始輸出和輸入的資料順序會是相同的。傳入*--sort-keys* 引數以按照鍵值的字母順序對輸出進行排序。

在 3.14 版的變更: The *json* module may now be directly executed as `python -m json`. For backwards compatibility, invoking the CLI as `python -m json.tool` remains supported.

命令列選項

infile

將被用於校驗或美化呈現的 JSON 文件：

```
$ python -m json mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

如果🄵有指定 *infile* 則會從*sys.stdin* 讀取輸入。

outfile

將 *infile* 的結果寫入到給定的 *outfile*。若未提供則寫入到*sys.stdout*。

--sort-keys

按照鍵值的字母順序對輸出字典進行排序。

在 3.5 版被加入.

--no-ensure-ascii

關閉非 ASCII 字元的自動跳🄵功能。詳情請參照*json.dumps()*。

在 3.9 版被加入.

--json-lines

將每一行輸入都單獨輸出🄵一個 JSON 物件。

在 3.8 版被加入.

--indent, --tab, --no-indent, --compact

互斥的空白字元控制選項。

在 3.9 版被加入.

-h, --help

顯示🄵明訊息。

🄵解

20.3 mailbox --- 以各種格式操作郵件信箱

原始碼： Lib/mailbox.py

This module defines two classes, *Mailbox* and *Message*, for accessing and manipulating on-disk mailboxes and the messages they contain. `Mailbox` offers a dictionary-like mapping from keys to messages. `Message` extends the *email.message* module's *Message* class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

也參考

***email* 模組**
Represent and manipulate messages.

20.3.1 Mailbox 物件

class mailbox.**Mailbox**

A mailbox, which may be inspected and modified.

The `Mailbox` class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from `Mailbox` and your code should instantiate a particular subclass.

The `Mailbox` interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the `Mailbox` instance with which they will be used and are only meaningful to that `Mailbox` instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a `Mailbox` instance using the set-like method *add()* and removed using a `del` statement or the set-like methods *remove()* and *discard()*.

`Mailbox` interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a *Message* instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a `Mailbox` instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the `Mailbox` instance.

The default `Mailbox` *iterator* iterates over message representations, not keys as the default `dictionary` iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

> **警告**
>
> Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is `Maildir`; try to avoid using single-file formats such as `mbox` for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods:

add(*message*)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

在 3.2 版的變更: Support for binary input was added.

remove(*key*)

__delitem__(*key*)

discard(*key*)

Delete the message corresponding to *key* from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)

Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()

Return an *iterator* over all keys

keys()

The same as `iterkeys()`, except that a `list` is returned rather than an *iterator*

itervalues()

__iter__()

Return an *iterator* over representations of all messages. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

> **備🄵**
>
> The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

values()

The same as `itervalues()`, except that a `list` is returned rather than an *iterator*

iteritems()

Return an *iterator* over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

items()

The same as `iteritems()`, except that a `list` of pairs is returned rather than an *iterator* of pairs.

get(*key*, *default=None*)

__getitem__(*key*)

Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)

Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(*key*)

Return a byte representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

在 3.2 版被加入.

get_string(*key*)

Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(*key*)

Return a *file-like* representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

在 3.2 版的變更: The file object really is a *binary file*; previously it was incorrectly returned in text mode. Also, the *file-like object* now supports the *context manager* protocol: you can use a `with` statement to automatically close it.

> **備🄵**
>
> Unlike other representations of messages, *file-like* representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)

Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()

Return a count of messages in the mailbox.

clear()

Delete all messages from the mailbox.

pop(*key*, *default=None*)

Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

popitem()

Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

update(*arg*)

Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using `__setitem__()`. As with `__setitem__()`, each *key* must already correspond to a message in the mailbox or else a `KeyError` exception will be raised, so in general it is incorrect for *arg* to be a `Mailbox` instance.

> **備🄵**
>
> Unlike with dictionaries, keyword arguments are not supported.

flush()

Write any pending changes to the filesystem. For some `Mailbox` subclasses, changes are always written immediately and `flush()` does nothing, but you should still make a habit of calling this method.

lock()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An `ExternalClashError` is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()

Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some `Mailbox` subclasses, this method does nothing.

`Mailbox` 物件

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MaildirMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

If *create* is `True` and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by `Maildir` without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

colon

The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

在 3.13 版的變更: *Maildir* now ignores files with a leading dot.

`Maildir` instances have all of the methods of *Mailbox* in addition to the following:

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return a `Maildir` instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return a `Maildir` instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

get_flags(*key*)

Return as a string the flags that are set on the message corresponding to *key*. This is the same as `get_message(key).get_flags()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a *MaildirMessage* object, use its *get_flags()* method instead, because changes made by the message's *set_flags()*, *add_flag()* and *remove_flag()* methods are not reflected here until the mailbox's *__setitem__()* method is called.

在 3.13 版被加入.

set_flags(*key*, *flags*)

On the message corresponding to *key*, set the flags specified by *flags* and unset all others. Calling `some_mailbox.set_flags(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a *MaildirMessage* object, use its *set_flags()* method instead, because changes made with this mailbox method will not be visible to the message object's method, *get_flags()*.

在 3.13 版被加入.

add_flag(*key*, *flag*)

On the message corresponding to *key*, set the flags specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's *add_flag()* method are similar to those for *set_flags()*; see the discussion there.

在 3.13 版被加入.

remove_flag(*key*, *flag*)

On the message corresponding to *key*, unset the flags specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's *remove_flag()* method are similar to those for *set_flags()*; see the discussion there.

在 3.13 版被加入.

get_info(*key*)

Return a string containing the info for the message corresponding to *key*. This is the same as `get_message(key).get_info()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a *MaildirMessage* object, use its *get_info()* method instead, because changes made by the message's *set_info()* method are not reflected here until the mailbox's *__setitem__()* method is called.

在 3.13 版被加入.

set_info(*key*, *info*)

Set the info of the message corresponding to *key* to *info*. Calling `some_mailbox.set_info(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a *MaildirMessage* object, use its *set_info()* method instead, because changes made with this mailbox method will not be visible to the message object's method, *get_info()*.

在 3.13 版被加入.

Some *Mailbox* methods implemented by `Maildir` deserve special remarks:

add(*message*)

__setitem__(*key*, *message*)

update(*arg*)

> **⚠ 警告**
>
> These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()

unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

`Maildir` instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

也參考

maildir man page from Courier
A specification of the format. Describes a common extension for supporting folders.

Using maildir format
Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

`mbox` 物件

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *mboxMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, `mbox` implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some *Mailbox* methods implemented by `mbox` deserve special remarks:

get_bytes(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

get_file(*key*, *from_=False*)

Using the file after calling *flush()* or *close()* on the `mbox` instance may yield unpredictable results or raise an exception.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

get_string(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

也參考

mbox man page from tin
A specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad
An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats
A history of mbox variations.

MH 物件

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MHMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The `MH` class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by **mh** to store its state and configuration.

`MH` instances have all of the methods of *Mailbox* in addition to the following:

在 3.13 版的變更: Supported folders that don't contain a `.mh_sequences` file.

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return an `MH` instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return an `MH` instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()

Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)

Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()

Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

> **備🄵**
>
> Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by MH deserve special remarks:

remove(*key*)

__delitem__(*key*)

discard(*key*)

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()

All changes to MH mailboxes are immediately applied, so this method does nothing.

close()

MH instances do not keep any open files, so this method is equivalent to *unlock()*.

> **也參考**
>
> **nmh - Message Handling System**
> Home page of **nmh**, an updated version of the original **mh**.
>
> **MH & nmh: Email for Users & Programmers**
> A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

Babyl 物件

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *BabylMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

`Babyl` instances have all of the methods of *`Mailbox`* in addition to the following:

`get_labels`()

Return a list of the names of all user-defined labels used in the mailbox.

> **備🄵**
>
> The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *`Mailbox`* methods implemented by `Babyl` deserve special remarks:

`get_file`(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an *`io.BytesIO`* instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

`lock`()

`unlock`()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

> **也參考**
>
> **Format of Version 5 Babyl Files**
> A specification of the Babyl format.
>
> **Reading Mail with Rmail**
> The Rmail manual, with some information on Babyl semantics.

`MMDF` 物件

`class` mailbox.**`MMDF`**(*path*, *factory=None*, *create=True*)

A subclass of *`Mailbox`* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *`MMDFMessage`* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are ”From ”, but additional occurrences of ”From ” are not transformed to ”>From ” when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *`Mailbox`* methods implemented by `MMDF` deserve special remarks:

`get_bytes`(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix ”From ” line. If *from_* is False, the first line of the file is dropped.

`get_file`(*key*, *from_=False*)

Using the file after calling *`flush()`* or *`close()`* on the `MMDF` instance may yield unpredictable results or raise an exception.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix ”From ” line. If *from_* is False, the first line of the file is dropped.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

也參考

mmdf man page from tin
A specification of MMDF format from the documentation of tin, a newsreader.

MMDF
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

20.3.2 `Message` 物件

class mailbox.**Message**(*message=None*)

A subclass of the *email.message* module's *Message*. Subclasses of `mailbox.Message` add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a `Message` instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that `Message` instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate `Message` representations might not be acceptable. For such situations, `Mailbox` instances also offer string and file-like representations, and a custom message factory may be specified when a `Mailbox` instance is initialized.

`MaildirMessage` 物件

class mailbox.**MaildirMessage**(*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	含義	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

`MaildirMessage` instances offer the following methods:

get_subdir()

Return either ”new” (if the message should be stored in the `new` subdirectory) or ”cur” (if the message should be stored in the `cur` subdirectory).

> **備𝔽**
>
> A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

set_subdir(*subdir*)

Set the subdirectory the message should be stored in. Parameter *subdir* must be either ”new” or ”cur”.

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if ”info” contains experimental semantics.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current ”info” is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If ”info” contains experimental information rather than flags, the current ”info” is not modified.

get_date()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)

Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()

Return a string containing the ”info” for a message. This is useful for accessing and modifying ”info” that is experimental (i.e., not a list of flags).

set_info(*info*)

Set ”info” to *info*, which should be a string.

When a `MaildirMessage` instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
”cur” subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a `MaildirMessage` instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
”cur” subdirectory	”unseen” sequence
”cur” subdirectory and S flag	no ”unseen” sequence
F flag	”flagged” sequence
R flag	”replied” sequence

When a `MaildirMessage` instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
”cur” subdirectory	”unseen” label
”cur” subdirectory and S flag	no ”unseen” label
P flag	”forwarded” or ”resent” label
R flag	”answered” label
T flag	”deleted” label

`mboxMessage` 物件

class mailbox.**mboxMessage**(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender's envelope address and the time of delivery are typically stored in a line beginning with ”From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	含義	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The ”R” and ”O” flags are stored in the *Status* header, and the ”D”, ”F”, and ”A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`mboxMessage` instances offer the following methods:

get_from()

Return a string representing the ”From ” line that marks the start of the message in an mbox mailbox. The leading ”From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the ”From ” line to *from_*, which should be specified without a leading ”From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `mboxMessage` instance is created based upon a *`MaildirMessage`* instance, a ”From ” line is generated based upon the *`MaildirMessage`* instance's delivery date, and the following conversions take place:

Resulting state	*`MaildirMessage`* state
R flag	S flag
O flag	”cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `mboxMessage` instance is created based upon an *`MHMessage`* instance, the following conversions take place:

Resulting state	*`MHMessage`* state
R flag and O flag	no ”unseen” sequence
O flag	”unseen” sequence
F flag	”flagged” sequence
A flag	”replied” sequence

When an `mboxMessage` instance is created based upon a *`BabylMessage`* instance, the following conversions take place:

Resulting state	*`BabylMessage`* state
R flag and O flag	no ”unseen” label
O flag	”unseen” label
D flag	”deleted” label
A flag	”answered” label

When a `mboxMessage` instance is created based upon an *`MMDFMessage`* instance, the ”From ” line is copied and all flags directly correspond:

Resulting state	*`MMDFMessage`* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage 物件

class mailbox.**MHMessage**(*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods:

get_sequences()

Return a list of the names of sequences that include this message.

set_sequences(*sequences*)

Set the list of sequences that include this message.

add_sequence(*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an MHMessage instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
”unseen” sequence	no S flag
”replied” sequence	R flag
”flagged” sequence	F flag

When an MHMessage instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
”unseen” sequence	no R flag
”replied” sequence	A flag
”flagged” sequence	F flag

When an MHMessage instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
”unseen” sequence	”unseen” label
”replied” sequence	”answered” label

`BabylMessage` 物件

class mailbox.**BabylMessage**(*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanation
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

get_labels()

Return a list of labels on the message.

set_labels(*labels*)

Set the list of labels on the message to *labels*.

add_label(*label*)

Add *label* to the list of labels on the message.

remove_label(*label*)

Remove *label* from the list of labels on the message.

get_visible()

Return a `Message` instance whose headers are the message's visible headers and whose body is empty.

set_visible(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode).

update_visible()

When a `BabylMessage` instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of `Date`, `From`, `Reply-To`, `To`, `CC`, and `Subject` that are present in the original headers but not the visible headers are added to the visible headers.

When a `BabylMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place:

Resulting state	`MaildirMessage` state
"unseen" label	no S flag
"deleted" label	T flag
"answered" label	R flag
"forwarded" label	P flag

When a `BabylMessage` instance is created based upon an *`mboxMessage`* or *`MMDFMessage`* instance, the *`Status`* and *`X-Status`* headers are omitted and the following conversions take place:

Resulting state	*`mboxMessage`* or *`MMDFMessage`* state
”unseen” label	no R flag
”deleted” label	D flag
”answered” label	A flag

When a `BabylMessage` instance is created based upon an *`MHMessage`* instance, the following conversions take place:

Resulting state	*`MHMessage`* state
”unseen” label	”unseen” sequence
”answered” label	”replied” sequence

`MMDFMessage` 物件

class mailbox.**MMDFMessage**(*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *`Message`* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with ”From ”. Likewise, flags that indicate the state of the message are typically stored in *`Status`* and *`X-Status`* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	含義	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The ”R” and ”O” flags are stored in the *`Status`* header, and the ”D”, ”F”, and ”A” flags are stored in the *`X-Status`* header. The flags and headers typically appear in the order mentioned.

`MMDFMessage` instances offer the following methods, which are identical to those offered by *`mboxMessage`*:

get_from()

Return a string representing the ”From ” line that marks the start of the message in an mbox mailbox. The leading ”From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the ”From ” line to *from_*, which should be specified without a leading ”From ” or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *`time.struct_time`* instance, a tuple suitable for passing to *`time.strftime()`*, or `True` (to use *`time.gmtime()`*).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `MMDFMessage` instance is created based upon a *`MaildirMessage`* instance, a ”From ” line is generated based upon the *`MaildirMessage`* instance's delivery date, and the following conversions take place:

Resulting state	*`MaildirMessage`* state
R flag	S flag
O flag	”cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `MMDFMessage` instance is created based upon an *`MHMessage`* instance, the following conversions take place:

Resulting state	*`MHMessage`* state
R flag and O flag	no ”unseen” sequence
O flag	”unseen” sequence
F flag	”flagged” sequence
A flag	”replied” sequence

When an `MMDFMessage` instance is created based upon a *`BabylMessage`* instance, the following conversions take place:

Resulting state	*`BabylMessage`* state
R flag and O flag	no ”unseen” label
O flag	”unseen” label
D flag	”deleted” label
A flag	”answered” label

When an `MMDFMessage` instance is created based upon an *`mboxMessage`* instance, the ”From ” line is copied and all flags directly correspond:

Resulting state	*`mboxMessage`* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

20.3.3 例外

The following exception classes are defined in the `mailbox` module:

exception mailbox.**Error**

The based class for all other module-specific exceptions.

exception mailbox.**NoSuchMailboxError**

Raised when a mailbox is expected but is not found, such as when instantiating a *Mailbox* subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception mailbox.**NotEmptyError**

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.**ExternalClashError**

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely generated file name already exists.

exception mailbox.**FormatError**

Raised when the data in a file cannot be parsed, such as when an *MH* instance attempts to read a corrupted `.mh_sequences` file.

20.3.4 范例

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # 可能🄵 None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]
```

(繼續下一頁)

(繼續上一頁)

```
            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

20.4 `mimetypes` --- 將檔案名稱對映到 MIME 類型

原始碼： Lib/mimetypes.py

The *mimetypes* module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call *init()* if they rely on the information *init()* sets up.

`mimetypes.`**guess_type**(*url*, *strict=True*)

Guess the type of a file based on its filename, path or URL, given by *url*. URL can be a string or a *path-like object*.

The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding is `None` for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a *Content-Encoding* header, **not** as a *Content-Transfer-Encoding* header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. However, the behavior of this module also depends on the underlying operating system. Only file types recognized by the OS or explicitly registered with Python's internal database can be identified. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

在 3.8 版的變更: 新增 *url* 做🄵*path-like object* 的支援。

在 3.13 版之後被🄵用: Passing a file path instead of URL is *soft deprecated*. Use *guess_file_type()* for this.

`mimetypes.`**guess_file_type**(*path*, *, *strict=True*)

Guess the type of a file based on its path, given by *path*. Similar to the *guess_type()* function, but accepts a path instead of URL. Path can be a string, a bytes object or a *path-like object*.

在 3.13 版被加入.

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()* and *guess_file_type()*.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

mimetypes.**guess_extension**(*type*, *strict=True*)

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()* and *guess_file_type()*. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.**init**(*files=None*)

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from *knownfiles*; on Windows, the current registry settings are loaded. Each file named in *files* or *knownfiles* takes precedence over those named before it. Calling *init()* repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

If *files* is `None` the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

在 3.2 版的變更: Previously, Windows registry settings were ignored.

mimetypes.**read_mime_types**(*filename*)

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (`'.'`), to strings of the form `'type/subtype'`. If the file *filename* does not exist or cannot be read, `None` is returned.

mimetypes.**add_type**(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.**inited**

Flag indicating whether or not the global data structures have been initialized. This is set to `True` by *init()*.

mimetypes.**knownfiles**

List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

mimetypes.**suffix_map**

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

mimetypes.**encodings_map**

Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**

Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

模組的使用範例：

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.4.1 MimeTypes 物件

The *MimeTypes* class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the *mimetypes* module.

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the *read()* or *readfp()* methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded "on top" of the default database.

suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global *suffix_map* defined in the module.

encodings_map

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global *encodings_map* defined in the module.

types_map

Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

types_map_inv

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

guess_extension(*type*, *strict=True*)

Similar to the *guess_extension()* function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)

Similar to the *guess_type()* function, using the tables stored as part of the object.

guess_file_type(*path*, ***, *strict=True*)

Similar to the *guess_file_type()* function, using the tables stored as part of the object.

在 3.13 版被加入.

guess_all_extensions(*type*, *strict=True*)

Similar to the *guess_all_extensions()* function, using the tables stored as part of the object.

read(*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses *readfp()* to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)

Load MIME type information from the Windows registry.

可用性: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

在 3.2 版被加入.

add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. Valid extensions start with a '.' or are empty. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

Deprecated since version 3.14, will be removed in version 3.16: Invalid, undotted extensions will raise a *ValueError* in Python 3.16.

20.4.2 Command-line usage

The `mimetypes` module can be executed as a script from the command line.

```
python -m mimetypes [-h] [-e] [-l] type [type ...]
```

The following options are accepted:

-h

--help

Show the help message and exit.

-e

--extension

Guess extension instead of type.

-l

--lenient

Additionally search for some common, but non-standard types.

By default the script converts MIME types to file extensions. However, if `--extension` is specified, it converts file extensions to MIME types.

For each `type` entry, the script writes a line into the standard output stream. If an unknown type occurs, it writes an error message into the standard error stream and exits with the return code `1`.

20.4.3 Command-line example

Here are some examples of typical usage of the `mimetypes` command-line interface:

```
$ # get a MIME type by a file name
$ python -m mimetypes filename.png
type: image/png encoding: None

$ # get a MIME type by a URL
$ python -m mimetypes https://example.com/filename.txt
type: text/plain encoding: None

$ # get a complex MIME type
$ python -m mimetypes filename.tar.gz
type: application/x-tar encoding: gzip

$ # get a MIME type for a rare file extension
$ python -m mimetypes filename.pict
error: unknown extension of filename.pict

$ # now look in the extended database built into Python
$ python -m mimetypes --lenient filename.pict
type: image/pict encoding: None

$ # get a file extension by a MIME type
$ python -m mimetypes --extension text/javascript
.js

$ # get a file extension by a rare MIME type
$ python -m mimetypes --extension text/xul
error: unknown type text/xul

$ # now look in the extended database again
$ python -m mimetypes --extension --lenient text/xul
.xul

$ # try to feed an unknown file extension
$ python -m mimetypes filename.sh filename.nc filename.xxx filename.txt
type: application/x-sh encoding: None
type: application/x-netcdf encoding: None
error: unknown extension of filename.xxx

$ # try to feed an unknown MIME type
$ python -m mimetypes --extension audio/aac audio/opus audio/future audio/x-wav
.aac
.opus
error: unknown type audio/future
```

20.5 `base64` --- Base16、Base32、Base64、Base85 資料編碼

原始碼： Lib/base64.py

這個模組提供將二進位資料編碼成可顯示 ASCII 字元以及解碼回原始資料的功能。這包括了 **RFC 4648** 中指定的編碼（Base64、Base32 和 Base16）和非標準的*Base85 編碼*。

該模組提供了兩個介面。新介面支援將類位元組物件 編碼成 ASCII *bytes*，🄵將包含 ASCII 的類位元組物件或字串解碼🄵*bytes*。支援 **RFC 4648** 中定義的兩種 base-64 字母表（常見和 URL 安全 (URL-safe) 及檔案系統安全 (filesystem-safe) 字母表）。

舊版介面不支援從字串解碼，但它提供對檔案物件進行編碼和解碼的函式。它僅支援 Base64 標準字母表，🄵且按照 **RFC 2045** 每 76 個字元添加🄵行字元。請注意，如果你需要 **RFC 2045** 的支援，你可能會

需要*email* 函式庫。

在 3.3 版的變更: 新介面的解碼功能現在接受 ASCII-only 的 Unicode 字串。

在 3.4 版的變更: 任何類位元組物件現在被該模組中的所有編碼和解碼函式接受。新增了對 Ascii85/Base85 的支援。

20.5.1 RFC 4648 編碼

RFC 4648 編碼適合對二進位資料進行編碼來使得電子郵件、URL 或是 HTTP POST 🄵容等傳輸管道能🄵安全地傳遞資料。

base64.**b64encode**(*s*, *altchars=None*)

使用 Base64 對類位元組物件 *s* 進行編碼🄵回傳編碼過的*bytes*。

可選的 *altchars* 必須是一個長度🄵 2 的類位元組物件，用來指定替代的字母表，以替🄵 + 和 / 字元。這使得應用程式可以生成對 URL 或檔案系統安全的 Base64 字串。預設值🄵 `None`，即使用標準的 Base64 字母表。

如果 *altchars* 的長度不是 2，可以斷言或引發*ValueError*。如果 *altchars* 不是類位元組物件，則會引發*TypeError*。

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)

將經過 Base64 編碼的類位元組物件或 ASCII 字串 *s* 解碼，🄵回傳解碼後的*bytes*。

可選的 *altchars* 必須是長度🄵 2 的類位元組物件或 ASCII 字串，用於指定替代字母表，取代 + 和 / 字元。

如果 *s* 填充 (pad) 不正確，將引發*binascii.Error* 例外。

如果 *validate* 🄵 `False`（預設值），在 padding check（填充檢查）之前，不屬於標準 base-64 字母表和替代字母表的字元將被🄵🄵。如果 *validate* 🄵 `True`，輸入中的這些非字母表字元將導致引發*binascii.Error*。

有關嚴格的 base64 檢查的更多資訊，請參🄵*binascii.a2b_base64()*。

如果 *altchars* 的長度不是 2，可能會斷言或引發*ValueError*。

base64.**standard_b64encode**(*s*)

使用標準 Base64 字母表對類位元組物件 *s* 進行編碼，🄵回傳編碼後的*bytes*。

base64.**standard_b64decode**(*s*)

使用標準 Base64 字母表對類位元組物件或 ASCII 字串 *s* 進行解碼，🄵回傳解碼後的*bytes*。

base64.**urlsafe_b64encode**(*s*)

使用 URL 安全和檔案系統安全的字母表對類位元組物件 *s* 進行編碼，該字母表將標準 Base64 字母表中的 + 替🄵🄵 -，/ 替🄵🄵 _，🄵回傳編碼後的*bytes*。結果仍可能包含 =。

base64.**urlsafe_b64decode**(*s*)

使用 URL 安全和檔案系統安全字母表對類位元組物件或 ASCII 字串 *s* 進行解碼，該字母表將標準 Base64 字母表中的 + 替🄵🄵 -，/ 替🄵🄵 _，🄵回傳解碼後的*bytes*。

base64.**b32encode**(*s*)

使用 Base32 對類位元組物件 *s* 進行編碼，🄵回傳編碼後的*bytes*。

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)

解碼經過 Base32 編碼的類位元組物件或 ASCII 字串 *s*，🄵回傳解碼後的*bytes*。

可選的 *casefold* 是一個是否接受小寫字母表作🄵輸入的旗標。出於安全性考量，預設值🄵 `False`。

RFC 4648 允許將數字 0 選擇性地對應對映🄵字母 O，🄵且允許將數字 1 選擇性地對映🄵字母 I 或字母 L。當可選的引數 *map01* 不🄵 `None` 時，指定數字 1 應該對映🄵哪個字母（當 *map01* 不🄵 `None` 時，數字 0 總是對映🄵字母 O）。出於安全性考量，預設值🄵 `None`，因此不允許在輸入中使用數字 0 和 1。

如果 *s* 的填充不正確或輸入中存在非字母表字元，將引發*binascii.Error*。

base64.**b32hexencode**(*s*)

類似於*b32encode()*，但使用在 **RFC 4648** 中定義的擴展十六進位字母表 (Extended Hex Alphabet)。

在 3.10 版被加入.

base64.**b32hexdecode**(*s*, *casefold=False*)

類似於*b32encode()*，但使用在 **RFC 4648** 中定義的擴展十六進位字母表。

這個版本不允許將數字 0 對映🄵字母 O，以及將數字 1 對映🄵字母 I 或字母 L，所有這些字元都包含在擴展十六進位字母表中，🄵且不能互🄵使用。

在 3.10 版被加入.

base64.**b16encode**(*s*)

使用 Base16 對類位元組物件 *s* 進行編碼，🄵回傳編碼後的*bytes*。

base64.**b16decode**(*s*, *casefold=False*)

解碼經過 Base16 編碼的類位元組物件或 ASCII 字串 *s*，🄵回傳解碼後的*bytes*。

可選的 *casefold* 是一個是否接受小寫字母表作🄵輸入的旗標。出於安全性考量，預設值🄵 `False`。

如果 *s* 的填充不正確或輸入中存在非字母表字元，將引發*binascii.Error*。

20.5.2 Base85 編碼

Base85 encoding is not formally specified but rather a de facto standard, thus different systems perform the encoding differently.

The *a85encode()* and *b85encode()* functions in this module are two implementations of the de facto standard. You should call the function with the Base85 implementation used by the software you intend to work with.

The two functions present in this module differ in how they handle the following:

- Whether to include enclosing <~ and ~> markers
- Whether to include newline characters
- The set of ASCII characters used for encoding
- Handling of null bytes

Refer to the documentation of the individual functions for more information.

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

使用 Ascii85 對類位元組物件 *b* 進行編碼，🄵回傳編碼後的*bytes*。

foldspaces 是一個可選的旗標，它使用特殊的短序列'y' 來替代連續的 4 個空格 (ASCII 0x20)，這是由'btoa' 支援的功能。這個特性不被「標准」的 Ascii85 編碼所支援。

wrapcol 控制輸出是否應該包含🄵行字元 (`b'\n'`)。如果這個值不🄵零，每行輸出的長度將不超過這個字元長度（不包含後面的🄵行符號）。

pad 控制是否在編碼之前將輸入填充🄵 4 的倍數。請注意，`btoa` 實作始終會填充。

adobe 控制編碼的位元組序列前後是否加上 <~ 和 ~>，這是 Adobe 實作中使用的。

在 3.4 版被加入.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

解碼經過 Ascii85 編碼的類位元組物件或 ASCII 字串 *b*，🄵回傳解碼後的*bytes*。

foldspaces 是一個旗標，指定是否應該將短序列'y' 視🄵 4 個連續的空格 (ASCII 0x20) 的簡寫。這個功能不受「標准」Ascii85 編碼的支援。

adobe 控制輸入序列是否符合 Adobe Ascii85 格式（即前後加上 <~ 和 ~>）。

ignorechars 是一個包含要從輸入中忽略的字元的類位元組物件或 ASCII 字串。這只包含空格字元，預設情🄵下包含 ASCII 中的所有空格字元。

在 3.4 版被加入.

base64.**b85encode**(*b*, *pad=False*)

使用 Base85（例如，git 風格的二進位差🄵 (binary diff)）對類位元組物件 *b* 進行編碼，🄵回傳編碼後的*bytes*。

如果 *pad* 🄵 true，則在編碼之前，輸入將使用 `b'\0'` 進行填充，以使其長度🄵 4 的倍數。

在 3.4 版被加入.

base64.**b85decode**(*b*)

解碼經過 base85 編碼的類位元組物件或 ASCII 字串 *b*，🄵回傳解碼後的*bytes*。必要時會隱式移除填充。

在 3.4 版被加入.

base64.**z85encode**(*s*)

使用 Z85（如用於 ZeroMQ）對類位元組物件 *s* 進行編碼，🄵回傳編碼後的*bytes*。有關更多資訊，請參🄵 Z85 規格。

在 3.13 版被加入.

base64.**z85decode**(*s*)

解碼經 Z85 編碼的類位元組物件或 ASCII 字串 *s*，🄵回傳解碼後的*bytes*。有關更多資訊，請參🄵 Z85 規格。

在 3.13 版被加入.

20.5.3 舊版介面

base64.**decode**(*input*, *output*)

解碼二進位檔案 *input* 的🄵容，🄵將結果的二進位資料寫入 *output* 檔案。*input* 和 *output* 必須是檔案物件。*input* 將被讀取，直到 `input.readline()` 回傳一個空的 bytes 物件🄵止。

base64.**decodebytes**(*s*)

解碼必須包含一行或多行的 base64 編碼資料類位元組物件 *s*，🄵回傳解碼後的*bytes*。

在 3.1 版被加入.

base64.**encode**(*input*, *output*)

編碼二進位檔案 *input* 的🄵容，🄵將結果的 base64 編碼資料寫入 *output* 檔案。*input* 和 *output* 必須是檔案物件。*input* 將被讀取，直到 `input.read()` 回傳一個空的 bytes 物件🄵止。*encode()* 會在輸出的每 76 個位元組之後插入一個🄵行字元 (`b'\n'`)，🄵確保輸出始終以🄵行字元結尾，符合 **RFC 2045** (MIME) 的規定。

base64.**encodebytes**(*s*)

對包含任意二進位資料的類位元組物件 *s* 進行編碼，🄵回傳包含 base64 編碼資料*bytes*，在每 76 個輸出位元組後插入🄵行字元 (`b'\n'`)，🄵確保有尾隨🄵行字元，符合 **RFC 2045** (MIME) 的規定。

在 3.1 版被加入.

模組的一個範例用法：

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

20.5.4 安全性注意事項

RFC 4648（第 12 節）中添加了一個新的安全性考量部分；建議對部署到正式環境的任何程式碼進行安全性部分的審查。

也參考

***binascii* 模組**
支援模組中包含 ASCII 到二進位 (ASCII-to-binary) 和二進位到 ASCII (binary-to-ASCII) 的轉🄵功能。

RFC 1521 - MIME（多用途網際網路郵件擴展）第一部分：指定和描述網際網路主體格式的機制。
第 5.2 節，”Base64 Content-Transfer-Encoding”，提供了 base64 編碼的定義。

20.6 `binascii` --- 在二進位制和 ASCII 之間轉🄵

The *binascii* module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like *base64* instead. The *binascii* module contains low-level functions written in C for greater speed that are used by the higher-level modules.

備🄵

`a2b_*` functions accept Unicode strings containing only ASCII characters. Other functions only accept *bytes-like objects* (such as *bytes*, *bytearray* and other objects that support the buffer protocol).

在 3.3 版的變更: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

binascii 模組定義了以下函式：

`binascii.`**`a2b_uu`**(*string*)

Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

`binascii.`**`b2a_uu`**(*data*, *, *backtick=False*)

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45. If *backtick* is true, zeros are represented by `'`` '` instead of spaces.

在 3.7 版的變更: 新增 *backtick* 參數。

`binascii.`**`a2b_base64`**(*string*, /, *, *strict_mode=False*)

Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

If *strict_mode* is true, only valid base64 data will be converted. Invalid base64 data will raise *binascii.Error*.

Valid base64:

- Conforms to RFC 3548.
- Contains only characters from the base64 alphabet.
- Contains no excess data after padding (including excess padding, newlines, etc.).
- Does not start with a padding.

在 3.11 版的變更: 新增 *strict_mode* 參數。

binascii.**b2a_base64**(*data*, ***, *newline=True*)

Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char if *newline* is true. The output of this function conforms to **RFC 3548**.

在 3.6 版的變更: 新增 *newline* 參數。

binascii.**a2b_qp**(*data*, *header=False*)

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per **RFC 1522**. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.**crc_hqx**(*data*, *value*)

Compute a 16-bit CRC value of *data*, starting with *value* as the initial CRC, and return the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.**crc32**(*data*[, *value*])

Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

在 3.0 版的變更: The result is always unsigned.

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The returned bytes object is therefore twice as long as the length of *data*.

Similar functionality (but returning a text string) is also conveniently accessible using the *bytes.hex()* method.

If *sep* is specified, it must be a single character str or bytes object. It will be inserted in the output after every *bytes_per_sep* input bytes. Separator placement is counted from the right end of the output by default, if you wish to count from the left, supply a negative *bytes_per_sep* value.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

在 3.8 版的變更: 新增 *sep* 與 *bytes_per_sep* 參數。

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of `b2a_hex()`. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise an `Error` exception is raised.

Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the `bytes.fromhex()` class method.

exception binascii.**Error**

Exception raised on errors. These are usually programming errors.

exception binascii.**Incomplete**

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

也參考

***base64* 模組**
Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

***quopri* 模組**
Support for quoted-printable encoding used in MIME email messages.

20.7 quopri --- 編碼和解碼 MIME 可列印字元資料

原始碼： Lib/quopri.py

該模組根據 **RFC 1521**：「MIME（多功能網際網路郵件擴充）第一部分：指定和描述網際網路訊息正文格式的機制」中的定義來執行可列印字元 (quoted-printable) 傳輸編碼和解碼。可列印字元編碼是🄵不可列印字元相對較少的資料而設計的；如果存在許多此類字元（例如發送圖形檔案時），則透過`base64` 模組提供的 Base64 編碼方案會更加簡潔。

quopri.**decode**(*input*, *output*, *header=False*)

解碼 *input* 檔案的🄵容🄵將解碼後的二進位資料寫入 *output* 檔案。*input* 和 *output* 必須是二進位檔案物件。如果可選參數 *header* 存在且🄵 true，則底🄵將被解碼🄵空格。這用於解碼如 **RFC 1522**：「MIME（多功能網際網路郵件擴充）第二部分：非 ASCII 文字的訊息標頭擴充」中所述的"Q" 編碼標頭。

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)

對 *input* 檔案的🄵容進行編碼，🄵將生成的可列印字元資料寫入 *output* 檔案。*input* 和 *output* 必須是二進位檔案物件。*quotetabs*，一個非可選旗標，控制是否對嵌入的空格和🄵表符號 (tab) 進行編碼；當🄵 true 時，它將對此類嵌入的空白進行編碼，當🄵 false 時，它將不對它們進行編碼。請注意，出現在列尾的空格和🄵表符號都會按照 **RFC 1521** 進行編碼。*header* 是一個旗標，用於控制空格是否按照 **RFC 1522** 編碼🄵底🄵。

quopri.**decodestring**(*s*, *header=False*)

與`decode()` 類似，不同之處在於它接受來源的`bytes` 🄵回傳相應的已解碼`bytes`。

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)

與`encode()` 類似，不同之處在於它接受來源的`bytes` 🄵回傳相應的已編碼`bytes`。預設情🄵下，它向`encode()` 函式的 *quotetabs* 參數發送一個 `False` 值。

也參考

***base64* 模組**
對 MIME Base64 資料進行編碼和解碼

CHAPTER 21

結構化標記處理工具

Python 支援多種模組來處理各種形式的結構化資料標記。這包括用於處理標準通用標記語言 (SGML, Standard Generalized Markup Language) 和超文本標記語言 (HTML, Hypertext Markup Language) 的模組，以及用於處理可擴展標記語言 (XML, Extensible Markup Language) 的幾個介面。

21.1 `html` --- 超文本標記語言 (HTML) 支援

原始碼： Lib/html/__init__.py

此模組定義了操作 HTML 的工具。

`html.`**`escape`**(*s*, *quote=True*)

將字串 *s* 中的 `&`、< 和 > 字元轉🄵🄵在 HTML 中安全的序列 (sequence)。如果你需要在 HTML 中顯示可能包含這些字元的文字，可以使用這個函式。如果選擇性的旗標 *quote* 🄵 true（預設），則 (") 與 (') 字元也會被轉🄵；這樣能包含在 HTML 中，被引號分隔的屬性值，如 `<a href="...">` 。如果 *quote* 設🄵 false，則 (") 與 (') 字元不會被轉🄵。

在 3.2 版被加入.

`html.`**`unescape`**(*s*)

將字串 *s* 中所有附名 (named) 且🄵數值的 (numeric) 字元參照（如：`>`、`>`、`>` ）轉🄵🄵對應的 Unicode 字元。此函式針對有效及無效的字元參照，皆使用 HTML 5 標準所定義的規則，以及*HTML 5 附名字元參照清單* 。

在 3.4 版被加入.

`html` 套件中的子模組🄵:

- *`html.parser`* -- 帶有寬松剖析模式的 HTML/XHTML 剖析器
- *`html.entities`* -- HTML 實體的定義

21.2 `html.parser` --- 簡單的 HTML 和 XHTML 剖析器

原始碼： Lib/html/parser.py

該模組定義了一個類🄴*HTMLParser*，是剖析 (parse) HTML（HyperText Mark-up Language、超文本標記語言）和 XHTML 格式文本檔案的基礎。

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

建立一個能🄵剖析無效標記的剖析器實例。

如果 *convert_charrefs* 🄵 `True`（預設值），所有字元參照 (reference)（`script`/`style` 元素中的參照除外）將自動轉🄵🄵相應的 Unicode 字元。

HTMLParser 實例被提供 HTML 資料，🄴在遇到開始標🄴、結束標🄴、文本、🄴解和其他標記元素時呼叫處理程式 (handler) 方法。使用者應該繼承*HTMLParser* 🄴覆蓋其方法以實作所需的行🄴。

此剖析器不檢查結束標🄴是否與開始標🄴匹配，也不會🄴透過結束外部元素來隱晦地被結束的元素呼叫結束標🄴處理程式。

在 3.4 版的變更: 新增關鍵字引數 *convert_charrefs*。

在 3.5 版的變更: 引數 *convert_charrefs* 的預設值現在是 `True`。

21.2.1 HTML 剖析器應用程式范例

以下的基礎範例是一個簡單的 HTML 剖析器，它使用*HTMLParser* 類🄴，當遇到開始標🄴、結束標🄴和資料時將它們印出：

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

輸出將是：

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

21.2.2 HTMLParser 方法

HTMLParser 實例具有以下方法：

HTMLParser.**feed**(*data*)

向剖析器提供一些文本。只要它由完整的元素組成，它就會被處理；不完整的資料會被緩衝，直到輸入更多資料或呼叫*close()*。*data* 必須是*str*。

HTMLParser.**close**()

F制處理所有緩衝資料，如同它後面跟有文件結束標F一樣。此方法可能有被衍生類F重新定義，以在輸入末尾定義額外的處理，但重新定義的版本仍應要呼叫*HTMLParser* 基底類F方法*close()*。

HTMLParser.**reset**()

重置實例。FF所有未處理的資料。這在實例化時被會隱晦地呼叫。

HTMLParser.**getpos**()

回傳目前列號 (line number) 和偏移量 (offset)。

HTMLParser.**get_starttag_text**()

回傳最近開F (open) 的開始標F的文本。這對於結構化處理通常不必要，但在處理「已部署」的 HTML 或以最少的更改重新生成輸入（可以保留屬性之間的空白等）時可能很有用。

當遇到資料或標記元素時將呼叫以下方法，F且它們應在子類F中被覆蓋。基底類F實作什F都不做（除了*handle_startendtag()*）：

HTMLParser.**handle_starttag**(*tag*, *attrs*)

呼叫此方法來處理元素的開始標F（例如 `<div id="main">`）。

tag 引數是轉FF小寫的標F名稱。*attrs* 引數是一個 `(name, value)` 對的列表，包含在標F的 `<>` 括號F找到的屬性。*name* 將被轉成小寫，*value* 中的引號會被F除，字元和實體參照也會被替F。

例如，對於標F `<A HREF="https://www.cwi.nl/">`，這個方法會以 `handle_starttag('a', [('href', 'https://www.cwi.nl/')])` 的形式被呼叫。

在屬性值中來自*html.entities* 的所有實體參照都會被替F。

HTMLParser.**handle_endtag**(*tag*)

呼叫此方法來處理元素的結束標F（例如 `</div>`）。

tag 引數是轉FF小寫的標F名稱。

HTMLParser.**handle_startendtag**(*tag*, *attrs*)

與*handle_starttag()* 類似，但在剖析器遇到 XHTML 樣式的空標F (`<img ... />`) 時呼叫。這個方法可能被需要這個特定詞F資訊 (lexical information) 的子類F覆蓋；預設實作只是呼叫*handle_starttag()* 和*handle_endtag()*。

HTMLParser.**handle_data**(*data*)

呼叫此方法來處理任意資料（例如文本節點與 `<script>...</script>` 和 `<style>...</style>` 的F容）。

HTMLParser.**handle_entityref**(*name*)

呼叫此方法來處理形式F `&name;`（例如 `>`）的附名字元參照，其中 *name* 是一般實體參照（例如 `'gt'`）。如果 *convert_charrefs* F `True`，則永遠不會呼叫此方法。

HTMLParser.**handle_charref**(*name*)

呼叫此方法來處理 `&#`*NNN*`;` 和 `&#x`*NNN*`;` 形式的十進位和十六進位數字字元參照。例如，`>` 的十進位等效F `>`，而十六進位F `>`；在這種情F下，該方法將收到 `'62'` 或 `'x3E'`。如果 *convert_charrefs* F `True`，則永遠不會呼叫此方法。

HTMLParser.**handle_comment**(*data*)

當遇到F解時呼叫此方法（例如 `<!--comment-->`）。

舉例來F，F解 `<!-- comment -->` 會使得此方法被以引數 `' comment '` 來呼叫。

Internet Explorer 條件式🄵解（conditional comments, condcoms）的🄵容也會被發送到這個方法，故以 `<!--[if IE 9]>IE9-specific content<![endif]-->` 🄵例，這個方法將會收到 `'[if IE 9]>IE9-specific content<![endif]'`。

`HTMLParser.`**`handle_decl`**(*decl*)

呼叫此方法來處理 HTML 文件類型聲明 (doctype declaration)（例如 `<!DOCTYPE html>`）。

decl 參數將是 `<!...>` 標記🄵聲明部分的全部🄵容（例如 `'DOCTYPE html'`）。

`HTMLParser.`**`handle_pi`**(*data*)

遇到處理指示 (processing instruction) 時會呼叫的方法。*data* 參數將包含整個處理指示。例如，對於處理指示 `<?proc color='red'>`，這個方法將以 `handle_pi("proc color='red'")` 形式被呼叫。它旨在被衍生類🄵覆蓋；基底類🄵實作中什🄵都不做。

> **備🄵**
>
> *`HTMLParser`* 類🄵使用 SGML 語法規則來處理指示。使用有 `?` 跟隨在後面的 XHTML 處理指示將導致 `?` 被包含在 *data* 中。

`HTMLParser.`**`unknown_decl`**(*data*)

當剖析器讀取無法識🄵的聲明時會呼叫此方法。

data 參數將是 `<![...]>` 標記🄵聲明的全部🄵容。有時被衍生類🄵被覆蓋會是好用的。在基底類🄵實作中什🄵都不做。

21.2.3 范例

以下類🄵實作了一個剖析器，將用於解🄵更多範例：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)
```

(繼續下一頁)

(繼續上一頁)

```
parser = MyHTMLParser()
```

剖析文件類型：

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
↪strict.dtd"
```

剖析一個具有一些屬性和標題的元素：

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

`script` 和 `style` 元素的🄴容按原樣回傳，無需進一步剖析：

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

剖析🄴解：

```
>>> parser.feed('<!--a comment-->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  : a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

剖析附名 (named) 且🄴數值的 (numeric) 字元參照，🄴將它們轉🄵🄴正確的字元（注意：這 3 個參照都等同於 `'>'`）：

```
>>> parser = MyHTMLParser()
>>> parser.feed('&gt;&#62;&#x3E;')
Data     : >>>

>>> parser = MyHTMLParser(convert_charrefs=False)
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

將不完整的區塊提供給*`feed()`* 是可行的，但是*`handle_data()`* 可能會被多次呼叫（除非 *convert_charrefs* 設定🄴 `True`）：

```
>>> for chunk in ['<sp', 'an>buff', 'ered', ' text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     :  text
End tag  : span
```

也能🄵剖析無效的 HTML（例如未加引號的屬性）：

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

21.3 `html.entities` --- HTML 一般實體的定義

原始碼： Lib/html/entities.py

該 module（模組）定義了四個字典：*html5*、*name2codepoint*、*codepoint2name* 以及*entitydefs*。

`html.entities.`**`html5`**

將 HTML5 命名字元引用[1] 對映到同等 Unicode 字元的字典，例如 `html5['gt;'] == '>'`。請注意，後面的的分號包含在名稱中（例如 `'gt;'`），但有些名稱即使🄵有分號也會被此標準接受：在這種情🄵下，名稱可帶有或不帶有 `';'`。請見*html.unescape()*。

在 3.3 版被加入.

`html.entities.`**`entitydefs`**

將 XHTML 1.0 實體定義對映到 ISO Latin-1 中的替🄵文字的字典。

`html.entities.`**`name2codepoint`**

將 HTML4 實體名稱對映到 Unicode 程式點的字典。

`html.entities.`**`codepoint2name`**

將 Unicode 程式點對映到 HTML4 實體名稱的字典。

🄵解

21.4 XML 處理模組

原始碼： Lib/xml/

Python 處理 XML 的介面被歸類於 `xml` 套件中。

備🄵

如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

[1] 請見 https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

請務必注意*xml* 套件中的模組要求至少有一個可用的 SAX 相容 XML 剖析器。Expat 剖析器包含在 Python 中，所以總是可以使用*xml.parsers.expat* 模組。

xml.dom 和*xml.sax* 套件的文件🄵 DOM 和 SAX 介面的 Python 🄵結的定義。

以下是 XML 處理子模組：

- *xml.etree.ElementTree*：ElementTree API，一個簡單且輕量級的 XML 處理器
- *xml.dom*：DOM API 定義
- *xml.dom.minidom*：最小的 DOM 實作
- *xml.dom.pulldom*：支援建置部分 DOM 樹
- *xml.sax*：SAX2 基底類🄵和便利函式
- *xml.parsers.expat*：Expat 剖析器🄵結

21.4.1 XML 安全性

攻擊者可以濫用 XML 功能來執行阻斷服務攻擊 (denial of service attack)、存取本地檔案、生成與其他機器的網路連接或繞過防火🄵。

Expat versions lower than 2.6.0 may be vulnerable to "billion laughs", "quadratic blowup" and "large tokens". Python may be vulnerable if it uses such older versions of Expat as a system-provided library. Check `pyexpat.EXPAT_VERSION`.

xmlrpc 容易受到「解壓縮炸彈」攻擊。

十億笑聲 / 指數實體擴展
: 十億笑聲攻擊（也稱🄵指數實體擴展 (exponential entity expansion)）使用多層巢狀實體。每個實體多次引用另一個實體，最終的實體定義包含一個小字串。指數擴展會🄵生數 GB 的文本，🄵消耗大量記憶體和 CPU 時間。

二次爆炸實體擴展
: 二次爆炸攻擊類似於十億笑聲攻擊；它也濫用實體擴展。它不是巢狀實體，而是一遍又一遍地重🄵一個具有幾千個字元的大型實體。該攻擊不如指數擴展那🄵有效率，但它不會觸發那些用來防止深度巢狀實體的剖析器對策。

解壓縮炸彈 (decompression bomb)
: 解壓縮炸彈（又名 ZIP bomb）適用於所有可以剖析壓縮 XML 串流（例如 gzip 壓縮的 HTTP 串流或 LZMA 壓縮檔案）的 XML 函式庫。對於攻擊者來🄵，它可以將傳輸的資料量🄵少三個或更多數量級。

大型 token
: Expat 需要重新剖析未完成的 token；如果🄵有 Expat 2.6.0 中引入的保護，這可能會導致二次 runtime 而導致剖析 XML 的應用程式出現阻斷服務。此問題記🄵於 **CVE 2023-52425**。

21.5 `xml.etree.cElementTree` --- ElementTree XML API

原始碼： Lib/xml/etree/ElementTree.py

The *xml.etree.ElementTree* module implements a simple and efficient API for parsing and creating XML data.

在 3.3 版的變更: This module will use a fast implementation whenever available.

在 3.3 版之後被🄵用: `xml.etree.cElementTree` 模組已被🄵用。

備🄵

如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

21.5.1 教學

This is a short tutorial for using *xml.etree.ElementTree* (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - *ElementTree* represents the whole XML document as a tree, and *Element* represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the *ElementTree* level. Interactions with a single XML element and its sub-elements are done on the *Element* level.

剖析 XML

We'll be using the fictive `country_data.xml` XML document as the sample data for this section:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() parses XML from a string directly into an *Element*, which is the root element of the parsed tree. Other parsing functions may create an *ElementTree*. Check the documentation to be sure.

As an *Element*, `root` has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

備🄵

Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
mytag text= sometext more text
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at *iterparse()*. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Where *immediate* feedback through events is wanted, calling method *XMLPullParser.flush()* can help reduce delay; please make sure to study the related security notes.

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, *Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() finds only elements with a tag which are direct children of the current element. *Element.find()* finds the *first* child with a particular tag, and *Element.text* accesses the element's text content. *Element.get()* accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

改動 XML 檔案

ElementTree provides a simple way to build XML documents and write them to files. The *ElementTree.write()* method serves this purpose.

Once created, an *Element* object may be manipulated by directly changing its fields (such as *Element.text*), adding and modifying attributes (*Element.set()* method), as well as adding new children (for example with *Element.append()*).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

XML 現在看起來像這樣：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
```

(繼續下一頁)

(繼續上一頁)

```
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can remove elements using *Element.remove()*. Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     # 使用 root.findall() 來避免在遍歷時移除
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Note that concurrent modification while iterating can lead to problems, just like when iterating and modifying Python lists or dicts. Therefore, the example first collects all matching elements with `root.findall()`, and only then iterates over the list of matches.

XML 現在看起來像這樣:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

建立 XML 文件

The *SubElement()* function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix ”fictional” and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a *find()* or *findall()*:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

21.5.2 XPath 支援

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

范例

Here’s an example that demonstrates some of the XPath capabilities of the module. We’ll be using the `countrydata` XML document from the *Parsing XML* section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

For XML with namespaces, use the usual qualified `{namespace}tag` notation:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

支援的 XPath 語法

語法	意義
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`. `{namespace}*` selects all tags in the given namespace, `{*}spam` selects tags named `spam` in any (or no) namespace, and `{}*` only selects tags that are not in a namespace. 在 3.8 版的變更: 新增對星號萬用字元的支援。
`*`	Selects all child elements, including comments and processing instructions. For example, `*/egg` selects all grandchildren named `egg`.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	選擇所有具有給定屬性的元素。
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[@attrib!='value']`	Selects all elements for which the given attribute does not have the given value. The value cannot contain quotes. 在 3.10 版被加入.
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[.='text']`	Selects all elements whose complete text content, including descendants, equals the given `text`. 在 3.7 版被加入.
`[.!='text']`	Selects all elements whose complete text content, including descendants, does not equal the given `text`. 在 3.10 版被加入.
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[tag!='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, does not equal the given `text`. 在 3.10 版被加入.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

21.5.3 Reference

函式

`xml.etree.ElementTree.`**`canonicalize`**(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

C14N 2.0 transformation function.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduces the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace declarations, the ordering of attributes, and ignorable whitespace.

This function takes an XML data string (*xml_data*) or a file path or file-like object (*from_file*) as input, converts it to the canonical form, and writes it out using the *out* file(-like) object, if provided, or returns it as a text string if not. The output file receives text, not bytes. It should therefore be opened in text mode with `utf-8` encoding.

Typical uses:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

The configuration *options* are as follows:

- *with_comments*: set to true to include comments (default: false)
- ***strip_text*: set to true to strip whitespace before and after text content**
  (default: false)
- ***rewrite_prefixes*: set to true to replace namespace prefixes by "n{number}"**
  (default: false)
- ***qname_aware_tags*: a set of qname aware tag names in which prefixes**
  should be replaced in text content (default: empty)
- ***qname_aware_attrs*: a set of qname aware attribute names in which prefixes**
  should be replaced in text content (default: empty)
- *exclude_attrs*: a set of attribute names that should not be serialised
- *exclude_tags*: a set of tag names that should not be serialised

In the option list above, "a set" refers to any collection or iterable of strings, no ordering is expected.

在 3.8 版被加入.

xml.etree.ElementTree.**Comment**(*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into to the tree using one of the *Element* methods.

xml.etree.ElementTree.**dump**(*elem*)

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

在 3.8 版的變更: The *dump()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)

Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)

Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

在 3.2 版被加入.

xml.etree.ElementTree.**indent**(*tree*, *space=' '*, *level=0*)

Appends whitespace to the subtree to indent the tree visually. This can be used to generate pretty-printed XML output. *tree* can be an Element or ElementTree. *space* is the whitespace string that will be inserted for each indentation level, two space characters by default. For indenting partial subtrees inside of an already indented tree, pass the initial indentation level as *level*.

在 3.9 版被加入.

xml.etree.ElementTree.**iselement**(*element*)

Check if an object appears to be a valid element object. *element* is an element instance. Return `True` if this is an element object.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing `(event, elem)` pairs; it has a `root` attribute that references the root element of the resulting XML tree once *source* is fully read. The iterator has the `close()` method that closes the internal file object if *source* is a filename.

Note that while *iterparse()* builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see *XMLPullParser*.

> **備🄵**
>
> *iterparse()* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
>
> If you need a fully populated element, look for "end" events instead.

在 3.4 版之後被🄵用: *parser* 引數。

在 3.8 版的變更: 新增 *context* 與 *check_hostname* 事件。

在 3.13 版的變更: 新增 `close()` 方法。

xml.etree.ElementTree.**parse**(*source*, *parser=None*)

Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *ElementTree* instance.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that *XMLParser* skips over processing instructions in the input instead of creating PI objects for them. An *ElementTree* will only contain processing instruction nodes if they have been inserted into to the tree using one of the *Element* methods.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

在 3.2 版被加入.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.**tostring**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an *Element* instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns an (optionally) encoded string containing the XML data.

在 3.4 版的變更: 新增 *short_empty_elements* 參數。

在 3.8 版的變更: 新增 *xml_declaration* 與 *default_namespace* 參數。

在 3.8 版的變更: The *tostring()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an *Element* instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

在 3.2 版被加入.

在 3.4 版的變更: 新增 *short_empty_elements* 參數。

在 3.8 版的變更: 新增 *xml_declaration* 與 *default_namespace* 參數。

在 3.8 版的變更: The *tostringlist()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns a tuple containing an *Element* instance and a dictionary.

21.5.4 XInclude support

This module provides limited support for XInclude directives, via the *xml.etree.ElementInclude* helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

范例

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the *xml.etree.ElementTree* module:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to "xml". The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to "text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 Reference

函式

`xml.etree.ElementInclude.`**`default_loader`**(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an *Element* instance. If the parse mode is `"text"`, this is a string. If the loader fails, it can return `None` or raise an exception.

`xml.etree.ElementInclude.`**`include`**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)

This function expands XInclude directives in-place in tree pointed by *elem*. *elem* is either the root *Element* or an *ElementTree* instance to find such element. *loader* is an optional resource loader. If omitted, it defaults to *default_loader()*. If given, it should be a callable that implements the same interface as *default_loader()*. *base_url* is base URL of the original file, to resolve relative include file references.

max_depth is the maximum number of recursive inclusions. Limited to reduce the risk of malicious content explosion. Pass `None` to disable the limitation.

在 3.9 版的變更: 新增 *base_url* 與 *max_depth* 參數。

Element 物件

class `xml.etree.ElementTree.`**Element**(*tag*, *attrib={}*, ***extra*)

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag

A string identifying what kind of data this element represents (the element type, in other words).

text

tail

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the *tail* attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see *itertext()*, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib

A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)

Adds the element *subelement* to the end of this element's internal list of subelements. Raises *TypeError* if *subelement* is not an *Element*.

extend(*subelements*)

Appends *subelements* from an iterable of elements. Raises *TypeError* if a subelement is not an *Element*.

在 3.2 版被加入.

find(*match*, *namespaces=None*)

Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findall(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findtext(*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

insert(*index*, *subelement*)

Inserts *subelement* at the given position in this element. Raises *TypeError* if *subelement* is not an *Element*.

iter(*tag=None*)

Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

在 3.2 版被加入.

iterfind(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

在 3.2 版被加入.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

在 3.2 版被加入.

makeelement(*tag*, *attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the *SubElement()* factory function instead.

remove(*subelement*)

Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. In a future release of Python, all elements will test as `True` regardless of whether subelements exist. Instead, prefer explicit `len(elem)` or `elem is not None` tests.:

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

在 3.12 版的變更: Testing the truth value of an Element emits *DeprecationWarning*.

Prior to Python 3.8, the serialisation order of the XML attributes of elements was artificially made predictable by sorting the attributes by their name. Based on the now guaranteed ordering of dicts, this arbitrary reordering was removed in Python 3.8 to preserve the order in which attributes were originally parsed or created by user code.

In general, user code should try not to depend on a specific ordering of attributes, given that the XML Information Set explicitly excludes the attribute order from conveying information. Code should be prepared to deal with any ordering on input. In cases where deterministic XML output is required, e.g. for cryptographic signing or test data sets, canonical serialisation is available with the *canonicalize()* function.

In cases where canonical output is not applicable but a specific attribute order is still desirable on output, code should aim for creating the attributes directly in the desired order, to avoid perceptual mismatches for readers of the code. In cases where this is difficult to achieve, a recipe like the following can be applied prior to serialisation to enforce an order independently from the Element creation:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree 物件

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*, *namespaces=None*)

Same as *Element.find()*, starting at the root of the tree.

findall(*match*, *namespaces=None*)

Same as *Element.findall()*, starting at the root of the tree.

findtext(*match*, *default=None*, *namespaces=None*)

Same as *Element.findtext()*, starting at the root of the tree.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*, *namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

在 3.2 版被加入.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or *file object*. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns the section root element.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, ***, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a *file object* opened for writing. *encoding*[Page 1375, 1] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (`str`) or binary (`bytes`). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

在 3.4 版的變更: 新增 *short_empty_elements* 參數。

在 3.8 版的變更: The `write()` method now preserves the attribute order specified by the user.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName 物件

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder 物件

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, *, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Generic element structure builder. This builder converts a sequence of start, data, end, comment and pi method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format.

element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

The *comment_factory* and *pi_factory* functions, when given, should behave like the *Comment()* and *ProcessingInstruction()* functions to create comments and processing instructions. When not given, the default factories will be used. When *insert_comments* and/or *insert_pis* is true, comments/pis will be inserted into the tree if they appear within the root element (but not outside of it).

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an *Element* instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

comment(*text*)

Creates a comment with the given *text*. If `insert_comments` is true, this will also add it to the tree.

在 3.8 版被加入.

pi(*target*, *text*)

Creates a process instruction with the given *target* name and *text*. If `insert_pis` is true, this will also add it to the tree.

在 3.8 版被加入.

In addition, a custom *TreeBuilder* object can provide the following methods:

doctype(*name*, *pubid*, *system*)

Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default *TreeBuilder* class.

在 3.2 版被加入.

start_ns(*prefix*, *uri*)

Is called whenever the parser encounters a new namespace declaration, before the `start()` callback for the opening element that defines it. *prefix* is `''` for the default namespace and the declared namespace prefix name otherwise. *uri* is the namespace URI.

在 3.8 版被加入.

end_ns(*prefix*)

Is called after the `end()` callback of an element that declared a namespace prefix mapping, with the name of the *prefix* that went out of scope.

在 3.8 版被加入.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, *, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)

A C14N 2.0 writer. Arguments are the same as for the *canonicalize()* function. This class does not build a tree but translates the callback events directly into a serialised form using the *write* function.

在 3.8 版被加入.

XMLParser 物件

class xml.etree.ElementTree.**XMLParser**(*, *target=None*, *encoding=None*)

This class is the low-level building block of the module. It uses *xml.parsers.expat* for efficient, event-based parsing of XML. It can be fed XML data incrementally with the *feed()* method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard *TreeBuilder* is used. If *encoding*[Page 1375, 1] is given, the value overrides the encoding specified in the XML file.

在 3.8 版的變更: Parameters are now *keyword-only*. The *html* argument is no longer supported.

close()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

feed(*data*)

Feeds data to the parser. *data* is encoded data.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

在 3.13 版被加入.

XMLParser.feed() calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. For further supported callback methods, see the *TreeBuilder* class. *XMLParser.close()* calls *target*'s method `close()`. *XMLParser* can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
```

(繼續下一頁)

(繼續上一頁)

```
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser 物件

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported.

feed(*data*)

Feed the given bytes data to the parser.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

在 3.13 版被加入.

close()

Signal the parser that the data stream is terminated. Unlike *XMLParser.close()*, this method always returns *None*. Any events not yet retrieved when the parser is closed can still be read with *read_events()*.

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered *Element* object, or other context value as follows.

- `start`、`end`: 目前的 Element。
- `comment`, `pi`: the current comment / processing instruction
- `start-ns`: a tuple `(prefix, uri)` naming the declared namespace mapping.
- `end-ns`: *None* (this may change in a future version)

Events provided in a previous call to *read_events()* will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from *read_events()* will have unpredictable results.

> **備🄵**
>
> *XMLPullParser* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.
>
> If you need a fully populated element, look for "end" events instead.

在 3.4 版被加入.

在 3.8 版的變更: 新增 *context* 與 *check_hostname* 事件。

例外

class xml.etree.ElementTree.**ParseError**

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code

A numeric error code from the expat parser. See the documentation of *xml.parsers.expat* for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

🄵解

21.6 xml.dom --- Document 物件模型 API

原始碼： Lib/xml/dom/__init__.py

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or "levels" in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a `DOMImplementation` object class which provides access to `Document` creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing `Document` object. In Python, each DOM implementation will provide a function *getDOMImplementation()*.

DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section *Conformance* for a detailed discussion of mapping requirements.

也參考

Document Object Model (DOM) Level 2 Specification
The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification
The W3C recommendation for the DOM supported by *xml.dom.minidom*.

Python Language Mapping Specification
This specifies the mapping from OMG IDL to Python.

21.6.1 模組🄵容

The *xml.dom* contains the following functions:

xml.dom.**registerDOMImplementation**(*name*, *factory*)

Register the *factory* function with the name *name*. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)

Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an *ImportError*. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

xml.dom.**EMPTY_NAMESPACE**

The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the *namespaceURI* parameter to a namespaces-specific method.

xml.dom.**XML_NAMESPACE**

The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

xml.dom.**XMLNS_NAMESPACE**

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

xml.dom.**XHTML_NAMESPACE**

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

In addition, *xml.dom* contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM

implementations must provide those. The `Node` class provided as part of this module does provide the constants used for the `nodeType` attribute on concrete `Node` objects; they are located within the class rather than at the module level to conform with the DOM specifications.

21.6.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

Interface	Section	Purpose
`DOMImplementation`	*DOMImplementation 物件*	Interface to the underlying implementation.
`Node`	*Node Objects*	Base interface for most objects in a document.
`NodeList`	*NodeList 物件*	Interface for a sequence of nodes.
`DocumentType`	*DocumentType 物件*	Information about the declarations needed to process a document.
`Document`	*Document Objects*	Object which represents an entire document.
`Element`	*Element Objects*	Element nodes in the document hierarchy.
`Attr`	*Attr Objects*	Attribute value nodes on element nodes.
`Comment`	*Comment Objects*	Representation of comments in the source document.
`Text`	*Text and CDATASection Objects*	Nodes containing textual content from the document.
`ProcessingInstructi`	*ProcessingInstruction 物件*	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation 物件

The `DOMImplementation` interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new `Document` and `DocumentType` objects using the `DOMImplementation` as well.

`DOMImplementation.`**`hasFeature`**(*feature*, *version*)

Return `True` if the feature identified by the pair of strings *feature* and *version* is implemented.

`DOMImplementation.`**`createDocument`**(*namespaceUri*, *qualifiedName*, *doctype*)

Return a new `Document` object (the root of the DOM), with a child `Element` object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a `DocumentType` object created by *`createDocumentType()`*, or `None`. In the Python DOM API, the first two arguments can also be `None` in order to indicate that no `Element` child is to be created.

`DOMImplementation.`**`createDocumentType`**(*qualifiedName*, *publicId*, *systemId*)

Return a new `DocumentType` object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of `Node`.

`Node.`**`nodeType`**

An integer representing the node type. Symbolic constants for the types are on the `Node` object: `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`. This is a read-only attribute.

`Node.`**`parentNode`**

The parent of the current node, or `None` for the document node. The value is always a `Node` object or `None`.

For `Element` nodes, this will be the parent element, except for the root element, in which case it will be the `Document` object. For `Attr` nodes, this is always `None`. This is a read-only attribute.

`Node.`**`attributes`**

A `NamedNodeMap` of attribute objects. Only elements have actual values for this; others provide `None` for this attribute. This is a read-only attribute.

`Node.`**`previousSibling`**

The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be `None`. This is a read-only attribute.

`Node.`**`nextSibling`**

The node that immediately follows this one with the same parent. See also *`previousSibling`*. If this is the last child of the parent, this attribute will be `None`. This is a read-only attribute.

`Node.`**`childNodes`**

A list of nodes contained within this node. This is a read-only attribute.

`Node.`**`firstChild`**

The first child of the node, if there are any, or `None`. This is a read-only attribute.

`Node.`**`lastChild`**

The last child of the node, if there are any, or `None`. This is a read-only attribute.

`Node.`**`localName`**

The part of the `tagName` following the colon if there is one, else the entire `tagName`. The value is a string.

`Node.`**`prefix`**

The part of the `tagName` preceding the colon if there is one, else the empty string. The value is a string, or `None`.

`Node.`**`namespaceURI`**

The namespace associated with the element name. This will be a string or `None`. This is a read-only attribute.

`Node.`**`nodeName`**

This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the `tagName` property for elements or the `name` property for attributes. For all node types, the value of this attribute will be either a string or `None`. This is a read-only attribute.

`Node.`**`nodeValue`**

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with *`nodeName`*. The value is a string or `None`.

`Node.`**`hasAttributes`**`()`

Return `True` if the node has any attributes.

`Node.`**`hasChildNodes`**`()`

Return `True` if the node has any child nodes.

`Node.`**`isSameNode`**`(`*other*`)`

Return `True` if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

> **備🄵**
>
> This is based on a proposed DOM Level 3 API which is still in the "working draft" stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.**appendChild**(*newChild*)

Add a new child node to this node at the end of the list of children, returning *newChild*. If the node was already in the tree, it is removed first.

Node.**insertBefore**(*newChild*, *refChild*)

Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, *ValueError* is raised. *newChild* is returned. If *refChild* is None, it inserts *newChild* at the end of the children's list.

Node.**removeChild**(*oldChild*)

Remove a child node. *oldChild* must be a child of this node; if not, *ValueError* is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its unlink() method should be called.

Node.**replaceChild**(*newChild*, *oldChild*)

Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, *ValueError* is raised.

Node.**normalize**()

Join adjacent text nodes so that all stretches of text are stored as single Text instances. This simplifies processing text from a DOM tree for many applications.

Node.**cloneNode**(*deep*)

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

NodeList 物件

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.**item**(*i*)

Return the *i*'th item from the sequence, if there is one, or None. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.**length**

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__() and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the *len()* built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

DocumentType 物件

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

DocumentType.**publicId**

The public identifier for the external subset of the document type definition. This will be a string or None.

DocumentType.**systemId**

The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

DocumentType.**internalSubset**

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

DocumentType.**name**

The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType.**entities**

This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType.**notations**

This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

Document Objects

A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

Document.**documentElement**

The one and only root element of the document.

Document.**createElement**(*tagName*)

Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.**createElementNS**(*namespaceURI*, *tagName*)

Create and return a new element with a namespace. The *tagName* may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.**createTextNode**(*data*)

Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.**createComment**(*data*)

Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.**createProcessingInstruction**(*target*, *data*)

Create and return a processing instruction node containing the *target* and *data* passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document.**createAttribute**(*name*)

Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)

Create and return an attribute node with a namespace. The *tagName* may have a prefix. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.**getElementsByTagName**(*tagName*)

Search for all descendants (direct children, children's children, etc.) with a particular element type name.

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

Search for all descendants (direct children, children's children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

`Element` is a subclass of `Node`, so inherits all the attributes of that class.

`Element`.**tagName**

The element type name. In a namespace-using document it may have colons in it. The value is a string.

`Element`.**getElementsByTagName**(*tagName*)

Same as equivalent method in the `Document` class.

`Element`.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

Same as equivalent method in the `Document` class.

`Element`.**hasAttribute**(*name*)

Return `True` if the element has an attribute named by *name*.

`Element`.**hasAttributeNS**(*namespaceURI*, *localName*)

Return `True` if the element has an attribute named by *namespaceURI* and *localName*.

`Element`.**getAttribute**(*name*)

Return the value of the attribute named by *name* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

`Element`.**getAttributeNode**(*attrname*)

Return the `Attr` node for the attribute named by *attrname*.

`Element`.**getAttributeNS**(*namespaceURI*, *localName*)

Return the value of the attribute named by *namespaceURI* and *localName* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

`Element`.**getAttributeNodeNS**(*namespaceURI*, *localName*)

Return an attribute value as a node, given a *namespaceURI* and *localName*.

`Element`.**removeAttribute**(*name*)

Remove an attribute by name. If there is no matching attribute, a *`NotFoundErr`* is raised.

`Element`.**removeAttributeNode**(*oldAttr*)

Remove and return *oldAttr* from the attribute list, if present. If *oldAttr* is not present, *`NotFoundErr`* is raised.

`Element`.**removeAttributeNS**(*namespaceURI*, *localName*)

Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

`Element`.**setAttribute**(*name*, *value*)

Set an attribute value from a string.

`Element`.**setAttributeNode**(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the `name` attribute matches. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *`InuseAttributeErr`* will be raised.

`Element`.**setAttributeNodeNS**(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the `namespaceURI` and `localName` attributes match. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *`InuseAttributeErr`* will be raised.

`Element`.**setAttributeNS**(*namespaceURI*, *qname*, *value*)

Set an attribute value from a string, given a *namespaceURI* and a *qname*. Note that a qname is the whole attribute name. This is different than above.

Attr Objects

`Attr` inherits from `Node`, so inherits all its attributes.

`Attr.`**`name`**

The attribute name. In a namespace-using document it may include a colon.

`Attr.`**`localName`**

The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

`Attr.`**`prefix`**

The part of the name preceding the colon if there is one, else the empty string.

`Attr.`**`value`**

The text value of the attribute. This is a synonym for the `nodeValue` attribute.

NamedNodeMap 物件

`NamedNodeMap` does *not* inherit from `Node`.

`NamedNodeMap.`**`length`**

The length of the attribute list.

`NamedNodeMap.`**`item`**(*index*)

Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the `value` attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized `getAttribute*()` family of methods on the `Element` objects.

Comment Objects

`Comment` represents a comment in the XML document. It is a subclass of `Node`, but cannot have child nodes.

`Comment.`**`data`**

The content of the comment as a string. The attribute contains all characters between the leading `<!--` and trailing `-->`, but does not include them.

Text and CDATASection Objects

The `Text` interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in `CDATASection` objects. These two interfaces are identical, but provide different values for the `nodeType` attribute.

These interfaces extend the `Node` interface. They cannot have child nodes.

`Text.`**`data`**

The content of the text node as a string.

> **備🄵**
>
> The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

ProcessingInstruction 物件

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

ProcessingInstruction.**target**

The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

ProcessingInstruction.**data**

The content of the processing instruction following the first whitespace character.

例外

The DOM Level 2 recommendation defines a single exception, *DOMException*, and a number of constants that allow applications to determine what sort of error occurred. *DOMException* instances carry a *code* attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the *code* attribute.

exception xml.dom.**DOMException**

Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception xml.dom.**DomstringSizeErr**

Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception xml.dom.**HierarchyRequestErr**

Raised when an attempt is made to insert a node where the node type is not allowed.

exception xml.dom.**IndexSizeErr**

Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception xml.dom.**InuseAttributeErr**

Raised when an attempt is made to insert an `Attr` node that is already present elsewhere in the document.

exception xml.dom.**InvalidAccessErr**

Raised if a parameter or an operation is not supported on the underlying object.

exception xml.dom.**InvalidCharacterErr**

This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

exception xml.dom.**InvalidModificationErr**

Raised when an attempt is made to modify the type of a node.

exception xml.dom.**InvalidStateErr**

Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.**NamespaceErr**

If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

exception xml.dom.**NotFoundErr**

Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

exception xml.dom.**NotSupportedErr**

Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.**NoDataAllowedErr**

This is raised if data is specified for a node which does not support data.

exception xml.dom.**NoModificationAllowedErr**

Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.**SyntaxErr**

Raised when an invalid or illegal string is specified.

exception xml.dom.**WrongDocumentErr**

Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

常數	例外
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

21.6.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL Type	Python Type
boolean	bool 或 int
int	int
long int	int
unsigned int	int
DOMString	str 或 bytes
null	None

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
         attribute string anotherValue;
```

yields three accessor functions: a "get" method for `someValue` (`_get_someValue()`), and "get" and "set" methods for `anotherValue` (`_get_anotherValue()` and `_set_anotherValue()`). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is *not* required to work, and may raise an *AttributeError*.

The Python DOM API, however, *does* require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. "Set" accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being "live". The Python DOM API does not require implementations to enforce such requirements.

21.7 `xml.dom.minidom` --- 最小的 DOM 實作

原始碼： Lib/xml/dom/minidom.py

`xml.dom.minidom` is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the *`xml.etree.ElementTree`* module for their XML processing instead.

> **備🄵**
>
> 如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

DOM applications typically start by parsing some XML into a DOM. With *`xml.dom.minidom`*, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # 以檔名來剖析 XML 檔案

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # 剖析開🄵的檔案

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

`parse()` 函式可以接受一個檔案名稱或開🄵的檔案物件。

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)

Return a `Document` from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

如果你有一個字串中的 XML，你可以使用*`parseString()`* 函式來代替：

xml.dom.minidom.**parseString**(*string*, *parser=None*)

Return a `Document` that represents the *string*. This method creates an *`io.StringIO`* object for the string and passes that on to *`parse()`*.

這兩個函式都會回傳一個 `Document` 物件，表示文件的🄵容。

What the *`parse()`* and *`parseString()`* functions do is connect an XML parser with a "DOM builder" that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a `Document` by calling a method on a "DOM Implementation" object. You can get this object either by calling the `getDOMImplementation()` function in the *xml.dom* package or the *xml.dom.minidom* module. Once you have a `Document`, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the `documentElement` property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the `unlink()` method to encourage early cleanup of the now-unneeded objects. `unlink()` is an *xml.dom.minidom*-specific extension to the DOM API that renders the node and its descendants essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

也參考

Document Object Model (DOM) Level 1 Specification
W3C 對*xml.dom.minidom* DOM 支援的建議。

21.7.1 DOM 物件

The definition of the DOM API for Python is given as part of the *xml.dom* module documentation. This section lists the differences between the API and *xml.dom.minidom*.

`Node.`**`unlink`**`()`

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically unlink *dom* when the `with` block is exited:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # 使用 dom。
```

`Node.`**`writexml`**`(`*writer*, *indent=''*, *addindent=''*, *newl=''*, *encoding=None*, *standalone=None*`)`

Write XML to the writer object. The writer receives texts but not bytes as input, it should have a `write()` method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

For the `Document` node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Similarly, explicitly stating the *standalone* argument causes the standalone document declarations to be added to the prologue of the XML document. If the value is set to `True`, `standalone="yes"` is added, otherwise it is set to `"no"`. Not stating the argument will omit the declaration from the document.

在 3.8 版的變更: The *writexml()* method now preserves the attribute order specified by the user.

在 3.9 版的變更: 新增 *standalone* 參數。

Node.**toxml**(*encoding=None*, *standalone=None*)

Return a string or byte string containing the XML represented by the DOM node.

With an explicit *encoding*[1] argument, the result is a byte string in the specified encoding. With no *encoding* argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

The *standalone* argument behaves exactly as in *writexml()*.

在 3.8 版的變更: 現在*toxml()* 方法會保留使用者指定的屬性順序。

在 3.9 版的變更: 新增 *standalone* 參數。

Node.**toprettyxml**(*indent='\t'*, *newl='\n'*, *encoding=None*, *standalone=None*)

Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

The *encoding* argument behaves like the corresponding argument of *toxml()*.

The *standalone* argument behaves exactly as in *writexml()*.

在 3.8 版的變更: The *toprettyxml()* method now preserves the attribute order specified by the user.

在 3.9 版的變更: 新增 *standalone* 參數。

21.7.2 DOM 范例

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)
```

(繼續下一頁)

[1] The encoding name included in the XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not valid in an XML document's declaration, even though Python accepts it as an encoding name. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

(繼續上一頁)

```
def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")

def handleSlideTitle(title):
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

21.7.3 minidom 和 DOM 標準

The *xml.dom.minidom* module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. *xml.dom.minidom* supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.

- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.
- `DOMException` is currently not supported in *xml.dom.minidom*. Instead, *xml.dom.minidom* uses standard Python exceptions such as *TypeError* and *AttributeError*.
- `NodeList` objects are implemented using Python's built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in *xml.dom.minidom*:

- `DOMTimeStamp`
- `EntityReference`

Most of these reflect information in the XML document that is not of general utility to most DOM users.

21.8 `xml.dom.pulldom` --- 支援建置部分 DOM 樹

原始碼： Lib/xml/dom/pulldom.py

The *xml.dom.pulldom* module provides a "pull parser" which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling "events" from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

> **備🄵**
>
> 如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

在 3.7.1 版的變更: The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

範例：

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` 是一個常數，可以是以下其中一個：

- `START_ELEMENT`
- `END_ELEMENT`
- `COMMENT`

- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node is an object of type xml.dom.minidom.Document, xml.dom.minidom.Element or xml.dom.minidom.Text.

Since the document is treated as a "flat" stream of events, the document "tree" is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the *DOMEventStream.expandNode()* method and switch to DOM-related processing.

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)

xml.sax.handler.ContentHandler 的子類🄵。

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)

xml.sax.handler.ContentHandler 的子類🄵。

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)

Return a *DOMEventStream* from the given input. *stream_or_string* may be either a file name, or a file-like object. *parser*, if given, must be an *XMLReader* object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *parseString()* function instead:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)

回傳一個表示 (Unicode) *string* 的*DOMEventStream*。

xml.dom.pulldom.**default_bufsize**

Default value for the *bufsize* parameter to *parse()*.

The value of this variable can be changed before calling *parse()* and the new value will take effect.

21.8.1 DOMEventStream 物件

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

在 3.11 版的變更: 已移除對 __getitem__() 方法的支援。

getEvent()

Return a tuple containing *event* and the current *node* as xml.dom.minidom.Document if event equals START_DOCUMENT, xml.dom.minidom.Element if event equals START_ELEMENT or END_ELEMENT or xml.dom.minidom.Text if event equals CHARACTERS. The current node does not contain information about its children, unless *expandNode()* is called.

expandNode(*node*)

展開 *node* 的所有子節點到 *node*。範例:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # 以下陳述式只會印出 '<p/>'
        print(node.toxml())
        doc.expandNode(node)
```

(繼續下一頁)

(繼續上一頁)

```
        # 以下陳述式會印出包含所有子節點的節點 '<p>Some text <div>and more</div></
↪p>'
        print(node.toxml())
```

reset()

21.9 xml.sax --- SAX2 剖析器支援

原始碼： Lib/xml/sax/__init__.py

The *xml.sax* package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

> **備🄵**
>
> 如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

在 3.7.1 版的變更: The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method *setFeature()* on the parser object and argument *feature_external_ges*.

The convenience functions are:

xml.sax.**make_parser**(*parser_list=[]*)

Create and return a SAX *XMLReader* object. The first parser found will be used. If *parser_list* is provided, it must be an iterable of strings which name modules that have a function named create_parser(). Modules listed in *parser_list* will be used before modules in the default list of parsers.

在 3.8 版的變更: The *parser_list* argument can be any iterable, not just a list.

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

Create a SAX parser and use it to parse a document. The document, passed in as *filename_or_stream*, can be a filename or a file object. The *handler* parameter needs to be a SAX *ContentHandler* instance. If *error_handler* is given, it must be a SAX *ErrorHandler* instance; if omitted, *SAXParseException* will be raised on all errors. There is no return value; all work must be done by the *handler* passed in.

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

Similar to *parse()*, but parses from a buffer *string* received as a parameter. *string* must be a *str* instance or a *bytes-like object*.

在 3.5 版的變更: 新增*str* 實例的支援。

A typical SAX application uses three kinds of objects: readers, handlers and input sources. "Reader" in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The *InputSource*, *Locator*, Attributes, AttributesNS, and *XMLReader* interfaces are defined in the module *xml.sax.xmlreader*. The handler interfaces are defined in *xml.sax.handler*. For convenience, *InputSource* (which is often instantiated directly) and the handler classes are also available from *xml.sax*. These interfaces are described below.

In addition to these classes, *xml.sax* provides the following exception classes.

exception xml.sax.**SAXException**(*msg*, *exception=None*)

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the *ErrorHandler* interface receive instances of this exception, it is not required to actually raise the exception --- it is also useful as a container for information.

When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be `None` or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)

Subclass of *SAXException* raised on parse errors. Instances of this class are passed to the methods of the SAX *ErrorHandler* interface to provide information about the parse error. This class supports the SAX *Locator* interface as well as the *SAXException* interface.

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)

Subclass of *SAXException* raised when a SAX *XMLReader* is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)

Subclass of *SAXException* raised when a SAX *XMLReader* is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

也參考

SAX: The Simple API for XML
This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

***xml.sax.handler* 模組**
Definitions of the interfaces for application-provided objects.

***xml.sax.saxutils* 模組**
Convenience functions for use in SAX applications.

***xml.sax.xmlreader* 模組**
Definitions of the interfaces for parser-provided objects.

21.9.1 SAXException 物件

The *SAXException* exception class supports the following methods:

SAXException.**getMessage**()

Return a human-readable message describing the error condition.

SAXException.**getException**()

Return an encapsulated exception object, or `None`.

21.10 `xml.sax.handler` --- SAX 處理函式的基本類🄵

原始碼： Lib/xml/sax/handler.py

The SAX API defines five kinds of handlers: content handlers, DTD handlers, error handlers, entity resolvers and lexical handlers. Applications normally only need to implement those interfaces whose events they are interested in;

they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module *xml.sax.handler*, so that all methods get default implementations.

class xml.sax.handler.**ContentHandler**

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.**DTDHandler**

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.**EntityResolver**

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.**ErrorHandler**

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

class xml.sax.handler.**LexicalHandler**

Interface used by the parser to represent low frequency events which may not be of interest to many applications.

In addition to these classes, *xml.sax.handler* provides symbolic constants for the feature and property names.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"

true: Perform Namespace processing.

false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"

true: Report the original prefixed names and attributes used for Namespace declarations.

false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"

true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.

false: Names are not necessarily interned, although they may be (default).

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"

true: Report all validation errors (implies external-general-entities and external-parameter-entities).

false: Do not report validation errors.

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"

true: Include all external general (text) entities.

false: Do not include external general entities.

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**all_features**

List of all features.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.handler.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.**property_dom_node**

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"
data type: Bytes
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.**all_properties**

List of all known property names.

21.10.1 ContentHandler 物件

Users are expected to subclass *ContentHandler* to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.**setDocumentLocator**(*locator*)

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.**startDocument**()

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for *setDocumentLocator()*).

ContentHandler.**endDocument**()

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the *startPrefixMapping()* and *endPrefixMapping()* events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that *startPrefixMapping()* and *endPrefixMapping()* events are not guaranteed to be properly nested relative to each-other: all *startPrefixMapping()* events will occur before the corresponding *startElement()* event, and all *endPrefixMapping()* events will occur after the corresponding *endElement()* event, but their order is not guaranteed.

ContentHandler.**endPrefixMapping**(*prefix*)

End the scope of a prefix-URI mapping.

See *startPrefixMapping()* for details. This event will always occur after the corresponding *endElement()* event, but the order of *endPrefixMapping()* events is not otherwise guaranteed.

ContentHandler.**startElement**(*name*, *attrs*)

Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the *Attributes* interface containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

ContentHandler.**endElement**(*name*)

Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElement()* event.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)

Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the *AttributesNS* interface containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.**endElementNS**(*name*, *qname*)

Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElementNS()* method, likewise the *qname* parameter.

ContentHandler.**characters**(*content*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

> **備🄵**
>
> The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.**ignorableWhitespace**(*whitespace*)

Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.**processingInstruction**(*target*, *data*)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.**skippedEntity**(*name*)

Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

21.10.2 DTDHandler 物件

DTDHandler instances provide the following methods:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)

Handle a notation declaration event.

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)

Handle an unparsed entity declaration event.

21.10.3 EntityResolver 物件

EntityResolver.**resolveEntity**(*publicId*, *systemId*)

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

21.10.4 ErrorHandler 物件

Objects with this interface are used to receive error and warning information from the *XMLReader*. If you create an object that implements this interface, then register the object with your *XMLReader*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a *SAXParseException* as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

ErrorHandler.**error**(*exception*)

Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

ErrorHandler.**fatalError**(*exception*)

Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

ErrorHandler.**warning**(*exception*)

Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

21.10.5 LexicalHandler 物件

Optional SAX2 handler for lexical events.

This handler is used to obtain lexical information about an XML document. Lexical information includes information describing the document encoding used and XML comments embedded in the document, as well as section boundaries for the DTD and for any CDATA sections. The lexical handlers are used in the same manner as content handlers.

Set the LexicalHandler of an XMLReader by using the setProperty method with the property identifier `'http://xml.org/sax/properties/lexical-handler'`.

LexicalHandler.**comment**(*content*)

Reports a comment anywhere in the document (including the DTD and outside the document element).

LexicalHandler.**startDTD**(*name*, *public_id*, *system_id*)

Reports the start of the DTD declarations if the document has an associated DTD.

LexicalHandler.**endDTD**()

Reports the end of DTD declaration.

LexicalHandler.**startCDATA**()

Reports the start of a CDATA marked section.

The contents of the CDATA marked section will be reported through the characters handler.

LexicalHandler.**endCDATA**()

Reports the end of a CDATA marked section.

21.11 `xml.sax.saxutils` --- SAX 工具程式

原始碼： Lib/xml/sax/saxutils.py

模組*xml.sax.saxutils*包含了許多在建立 SAX 應用程式時非常有用的類🄵和函式，這些類🄵和函式可以直接使用，也可以作🄵基底類🄵。

xml.sax.saxutils.**escape**(*data*, *entities={}*)

在資料字串中跳🄴 `'&'`、`'<'` 及 `'>'`。

你可以透過傳入字典作🄵可選的 *entities* 參數，來跳🄴其他資料字串。鍵和值都必須是字串；每個鍵都會被對應的值取代。即使提供 *entities*，`'&'`、`'<'` 和 `'>'` 字元始終會被跳🄴。

> **備🄵**
>
> 這個函式只能用來跳🄴 XML 中無法直接使用的字元。請勿將此函式用作一般的字串轉🄵函式。

xml.sax.saxutils.**unescape**(*data*, *entities={}*)

取消跳🄵資料字串中的 `'&'`、`'<'` 和 `'>'`。

你可以透過傳入字典作🄵可選的 *entities* 參數，來取消其他字串的資料。鍵和值都必須是字串；每個鍵都會被對應的值取代。即使提供了 *entities*，`'&amp'`、`'<'` 和 `'>'` 總是會取消跳🄴。

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)

類似於*escape()*，但也准備 *data* 作🄵屬性值。*quoteattr()* 會根據 *data* 的🄵容選擇引號字元，嘗試避免在字串中編碼任何引號字元。如果 *data* 中已經有單引號和雙引號字元，雙引號字元將被編碼，而 *data* 將被雙引號包覆。結果字串可直接用作屬性值：

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

當使用參考具體語法🄵 HTML 或任何 SGML 🄵生屬性值時，此功能非常有用。

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

這個類🄵透過將 SAX 事件寫回 XML 文件來實作*ContentHandler* 介面。🄵句話🄵，使用*XMLGenerator* 作🄵🄵容處理器將會重現被剖析的原始文件。*out* 應該是類檔案物件，預設🄵 *sys.stdout*。*encoding* 是輸出串流的編碼，預設🄵 `'iso-8859-1'`。*short_empty_elements* 控制不包含🄵容的元素的格式：如果設定🄵 `False`（預設值），它們會以一對開始/結束（start/end）標🄵的形式輸出；如果設定🄵 `True`，它們會以單一自封（self-closed）標🄵的形式輸出。

在 3.2 版的變更: 新增 *short_empty_elements* 參數。

class xml.sax.saxutils.**XMLFilterBase**(*base*)

這個類🄵的設計是在*XMLReader* 與用🄵端應用程式的事件處理程式之間。預設情🄵下，它什🄵都不做，只是將要求傳送到🄵讀器，🄵將事件未經修改地傳送到處理程式，但子類🄵可以覆寫特定的方法，以在傳送它們的時候修改事件串流或配置請求。

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)

這個函式接收一個輸入來源和一個可選的基底 URL，🄵回傳一個完全解析的*InputSource* 物件，以准備讀取。輸入來源可以是字串、類檔案物件或*InputSource* 物件；剖析器會使用這個函式來實作它們的*parse()* 方法的多型 *source* 引數。

21.12 xml.sax.xmlreader --- XML 剖析器的介面

原始碼： Lib/xml/sax/xmlreader.py

SAX parsers implement the *XMLReader* interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by *xml.sax.make_parser()* with no arguments to create a new parser object.

class xml.sax.xmlreader.**XMLReader**

Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.**IncrementalParser**

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.**Locator**

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.**InputSource**(*system_id=None*)

Encapsulation of the information needed by the *XMLReader* to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the *XMLReader.parse()* method and for returning from EntityResolver.resolveEntity.

An *InputSource* belongs to the application, the *XMLReader* is not allowed to modify *InputSource* objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)

This is an implementation of the `Attributes` interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)

Namespace-aware variant of *AttributesImpl*, which will be passed to `startElementNS()`. It is derived from *AttributesImpl*, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the `AttributesNS` interface (see section *The AttributesNS Interface*).

21.12.1 XMLReader 物件

The *XMLReader* interface supports the following methods:

XMLReader.**parse**(*source*)

Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source -- typically a file name or a URL), a *pathlib.Path* or *path-like* object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

在 3.5 版的變更: 新增對字元串流的支援。

在 3.8 版的變更: 新增對類路徑物件的支援。

XMLReader.**getContentHandler**()

Return the current *ContentHandler*.

XMLReader.**setContentHandler**(*handler*)

Set the current *ContentHandler*. If no *ContentHandler* is set, content events will be discarded.

XMLReader.**getDTDHandler**()

Return the current *DTDHandler*.

XMLReader.**setDTDHandler**(*handler*)

Set the current *DTDHandler*. If no *DTDHandler* is set, DTD events will be discarded.

XMLReader.**getEntityResolver**()

Return the current *EntityResolver*.

XMLReader.**setEntityResolver**(*handler*)

Set the current *EntityResolver*. If no *EntityResolver* is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.**getErrorHandler**()

Return the current *ErrorHandler*.

XMLReader.**setErrorHandler**(*handler*)

Set the current error handler. If no *ErrorHandler* is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.**setLocale**(*locale*)

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.**getFeature**(*featurename*)

Return the current setting for feature *featurename*. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module *xml.sax.handler*.

XMLReader.**setFeature**(*featurename*, *value*)

Set the *featurename* to *value*. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

XMLReader.**getProperty**(*propertyname*)

Return the current setting for property *propertyname*. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module *xml.sax.handler*.

XMLReader.**setProperty**(*propertyname*, *value*)

Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

21.12.2 IncrementalParser 物件

Instances of *IncrementalParser* offer the following additional methods:

IncrementalParser.**feed**(*data*)

Process a chunk of *data*.

IncrementalParser.**close**()

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.**reset**()

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

21.12.3 Locator Objects

Instances of *Locator* provide these methods:

Locator.**getColumnNumber**()

Return the column number where the current event begins.

Locator.**getLineNumber**()

Return the line number where the current event begins.

Locator.**getPublicId**()

Return the public identifier for the current event.

Locator.**getSystemId**()

Return the system identifier for the current event.

21.12.4 InputSource 物件

InputSource.**setPublicId**(*id*)

Sets the public identifier of this *InputSource*.

InputSource.**getPublicId**()

Returns the public identifier of this *InputSource*.

InputSource.**setSystemId**(*id*)

Sets the system identifier of this *InputSource*.

InputSource.**getSystemId**()

Returns the system identifier of this *InputSource*.

InputSource.**setEncoding**(*encoding*)

Sets the character encoding of this *InputSource*.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the *InputSource* is ignored if the *InputSource* also contains a character stream.

InputSource.**getEncoding**()

Get the character encoding of this InputSource.

InputSource.**setByteStream**(*bytefile*)

Set the byte stream (a *binary file*) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

InputSource.**getByteStream**()

Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

InputSource.**setCharacterStream**(*charfile*)

Set the character stream (a *text file*) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

InputSource.**getCharacterStream**()

Get the character stream for this input source.

21.12.5 The `Attributes` Interface

`Attributes` objects implement a portion of the *mapping protocol*, including the methods `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, and `values()`. The following methods are also provided:

`Attributes.`**`getLength`**`()`

Return the number of attributes.

`Attributes.`**`getNames`**`()`

Return the names of the attributes.

`Attributes.`**`getType`**`(`*name*`)`

Returns the type of the attribute *name*, which is normally `'CDATA'`.

`Attributes.`**`getValue`**`(`*name*`)`

Return the value of attribute *name*.

21.12.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available:

`AttributesNS.`**`getValueByQName`**`(`*name*`)`

Return the value for a qualified name.

`AttributesNS.`**`getNameByQName`**`(`*name*`)`

Return the `(namespace, localname)` pair for a qualified *name*.

`AttributesNS.`**`getQNameByName`**`(`*name*`)`

Return the qualified name for a `(namespace, localname)` pair.

`AttributesNS.`**`getQNames`**`()`

Return the qualified names of all attributes.

21.13 `xml.parsers.expat` --- 使用 Expat 進行快速 XML 剖析

備🄵

如果你需要剖析不受信任或未經驗證的資料，請參🄵*XML 安全性*。

The *`xml.parsers.expat`* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

這個模組提供一個例外和一個型🄵物件：

`exception` `xml.parsers.expat.`**`ExpatError`**

The exception raised when Expat reports an error. See section *ExpatError 例外* for more information on interpreting Expat errors.

`exception` `xml.parsers.expat.`**`error`**

`ExpatError` 的🄵名。

xml.parsers.expat.**XMLParserType**

The type of the return values from the *ParserCreate()* function.

xml.parsers.expat 模組包含兩個函式：

xml.parsers.expat.**ErrorString**(*errno*)

回傳一個給定錯誤編號 *errno* 的解釋字串。

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)

Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler 將會收到每個元素的以下字串：

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

也參考

Expat XML 剖析器
Expat 專案的首頁。

21.13.1 XMLParser 物件

xmlparser 物件擁有以下方法：

xmlparser.**Parse**(*data*[, *isfinal*])

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

xmlparser.**ParseFile**(*file*)

Parse XML data reading from the object *file*. *file* only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

xmlparser.**SetBase**(*base*)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the *ExternalEntityRefHandler()*, *NotationDeclHandler()*, and *UnparsedEntityDeclHandler()* functions.

xmlparser.**GetBase**()

Returns a string containing the base set by a previous call to *SetBase()*, or None if *SetBase()* hasn't been called.

xmlparser.**GetInputContext**()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None.

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])

Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the *ExternalEntityRefHandler()* handler function, described below. The child parser is created with the *ordered_attributes* and *specified_attributes* set to the values of this parser.

xmlparser.**SetParamEntityParsing**(*flag*)

Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE and XML_PARAM_ENTITY_PARSING_ALWAYS. Return true if setting the flag was successful.

xmlparser.**UseForeignDTD**([*flag*])

Calling this with a true value for *flag* (the default) will cause Expat to call the *ExternalEntityRefHandler* with *None* for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the *ExternalEntityRefHandler* will still be called, but the *StartDoctypeDeclHandler* and *EndDoctypeDeclHandler* will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the *Parse()* or *ParseFile()* methods are called; calling it after either of those have been called causes *ExpatError* to be raised with the *code* attribute set to errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING].

xmlparser.**SetReparseDeferralEnabled**(*enabled*)

> **⚠ 警告**
>
> Calling SetReparseDeferralEnabled(False) has security implications, as detailed below; please make sure to understand these consequences prior to using the SetReparseDeferralEnabled method.

Expat 2.6.0 introduced a security mechanism called "reparse deferral" where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may —depending of the sizing of input chunks pushed to Expat —no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling SetReparseDeferralEnabled(False) disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling SetReparseDeferralEnabled(True) allows re-enabling reparse deferral.

Note that *SetReparseDeferralEnabled()* has been backported to some prior releases of CPython as a security fix. Check for availability of *SetReparseDeferralEnabled()* using *hasattr()* if used in code running across a variety of Python versions.

在 3.13 版被加入.

xmlparser.**GetReparseDeferralEnabled**()

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

在 3.13 版被加入.

xmlparser 物件擁有以下屬性:

xmlparser.**buffer_size**

The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

xmlparser.**buffer_text**

Setting this to true causes the xmlparser object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time. Note that when it is false, data that does not contain newlines may be chunked too.

xmlparser.**buffer_used**

If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

xmlparser.**ordered_attributes**

Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

xmlparser.**specified_attributes**

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an xmlparser object, and will only have correct values once a call to Parse() or ParseFile() has raised an *xml.parsers.expat.ExpatError* exception.

xmlparser.**ErrorByteIndex**

Byte index at which an error occurred.

xmlparser.**ErrorCode**

Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the errors object.

xmlparser.**ErrorColumnNumber**

Column number at which an error occurred.

xmlparser.**ErrorLineNumber**

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an xmlparser object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

xmlparser.**CurrentByteIndex**

Current byte index in the parser input.

xmlparser.**CurrentColumnNumber**

Current column number in the parser input.

xmlparser.**CurrentLineNumber**

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object *o*, use `o.handlername = func`. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings, and *standalone* will be `1` if the document is declared standalone, `0` if it is declared not to be standalone, or `-1` if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE ...`). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or `None` if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(*name*, *model*)

Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are `'CDATA'`, `'ID'`, `'IDREF'`, ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or `None` if there is no default value (`#IMPLIED` values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.**StartElementHandler**(*name*, *attributes*)

Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If *ordered_attributes* is true, this is a list (see *ordered_attributes* for a full description). Otherwise it's a dictionary mapping names to values.

xmlparser.**EndElementHandler**(*name*)

Called for the end of every element.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)

Called for every processing instruction.

xmlparser.**CharacterDataHandler**(*data*)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information. Note that the character data may be chunked even if it is short and so you may receive more than one call to *CharacterDataHandler()*. Set the *buffer_text* instance attribute to `True` to avoid that.

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be `None` for external entities. The *notationName* parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

`xmlparser.`**`NotationDeclHandler`**(*notationName*, *base*, *systemId*, *publicId*)

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

`xmlparser.`**`StartNamespaceDeclHandler`**(*prefix*, *uri*)

Called when an element contains a namespace declaration. Namespace declarations are processed before the *`StartElementHandler`* is called for the element on which declarations are placed.

`xmlparser.`**`EndNamespaceDeclHandler`**(*prefix*)

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the *`StartNamespaceDeclHandler`* was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding *`EndElementHandler`* for the end of the element.

`xmlparser.`**`CommentHandler`**(*data*)

Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

`xmlparser.`**`StartCdataSectionHandler`**()

Called at the start of a CDATA section. This and *`EndCdataSectionHandler`* are needed to be able to identify the syntactical start and end for CDATA sections.

`xmlparser.`**`EndCdataSectionHandler`**()

在 CDATA 區塊結束時呼叫。

`xmlparser.`**`DefaultHandler`**(*data*)

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

`xmlparser.`**`DefaultHandlerExpand`**(*data*)

This is the same as the *`DefaultHandler()`*, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

`xmlparser.`**`NotStandaloneHandler`**()

Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

`xmlparser.`**`ExternalEntityRefHandler`**(*context*, *base*, *systemId*, *publicId*)

Called for references to external entities. *base* is the current base, as set by a previous call to *`SetBase()`*. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *`DefaultHandler`* callback, if provided.

21.13.2 ExpatError 例外

`ExpatError` exceptions have a number of interesting attributes:

ExpatError.**code**

Expat's internal error number for the specific error. The *errors.messages* dictionary maps these error numbers to Expat's error messages. For example:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The *errors* module also provides error message constants and a dictionary *codes* mapping these messages back to the error codes, see below.

ExpatError.**lineno**

Line number on which the error was detected. The first line is numbered `1`.

ExpatError.**offset**

Character offset into the line where the error occurred. The first column is numbered `0`.

21.13.3 范例

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

這個程式的輸出是:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the *xml.parsers.expat.model* module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.**XML_CTYPE_ANY**

The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**

The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**

Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

xml.parsers.expat.model.**XML_CQUANT_NONE**

No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.**XML_CQUANT_OPT**

The model is optional: it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.**XML_CQUANT_PLUS**

The model must occur one or more times (like `A+`).

xml.parsers.expat.model.**XML_CQUANT_REP**

The model must occur zero or more times, as for `A*`.

21.13.5 Expat 錯誤常數

The following constants are provided in the *xml.parsers.expat.errors* module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its *code* attribute with `errors.codes[errors.XML_ERROR_`*CONSTANT_NAME*`]`.

`errors` 模組有以下屬性：

xml.parsers.expat.errors.**codes**

A dictionary mapping string descriptions to their error codes.

在 3.2 版被加入.

xml.parsers.expat.errors.**messages**

A dictionary mapping numeric error codes to their string descriptions.

在 3.2 版被加入.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**

An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**

A character reference referred to a character which is illegal in XML (for example, character `0`, or '`&#0;`').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**

An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**

An attribute was used more than once in a start tag.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**

Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value `0`) in a UTF-8 input stream.

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**

Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**

An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**

The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**

Expat was not able to allocate memory internally.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**

A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**

An incomplete character was found in the input.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**

Some unspecified syntax error was encountered.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**

An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**

Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**

A reference was made to an entity which was not defined.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**

The document encoding is not supported by Expat.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**

A CDATA marked section was not closed.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**

The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**

An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**

A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**

An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**

The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**

The document contained no document element at all.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**

Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**

The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**

An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.**XML_ERROR_ABORTED**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_FINISHED**

The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XML**

An attempt was made to undeclare reserved namespace prefix `xml` or to bind it to another namespace URI.

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XMLNS**

An attempt was made to declare or undeclare reserved namespace prefix `xmlns`.

xml.parsers.expat.errors.**XML_ERROR_RESERVED_NAMESPACE_URI**

An attempt was made to bind the URI of one the reserved namespace prefixes `xml` and `xmlns` to another namespace prefix.

xml.parsers.expat.errors.**XML_ERROR_INVALID_ARGUMENT**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_NO_BUFFER**

This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_AMPLIFICATION_LIMIT_BREACH**

The limit on input amplification factor (from DTD and entities) has been breached.

xml.parsers.expat.errors.**XML_ERROR_NOT_STARTED**

The parser was tried to be stopped or suspended before it started.

在 3.14 版被加入.

🄵解

CHAPTER 22

網路協定 (Internet protocols) 及支援

這個章節講述的模組實作了網路協定及相關技術的支援；他們全都是用 Python 實作的。這🄵的大多數模組都需要相依於系統的模組*socket*，目前普遍的平台都支援該模組。以下🄵概述：

22.1 webbrowser --- 方便的網頁🄵覽器控制器

原始碼： Lib/webbrowser.py

webbrowser 模組提供了一個高階介面，允許向使用者顯示基於網頁的文件。在大多數情🄵下，只需從此模組呼叫*open()* 函式即可完成正確的操作。

在 Unix 下，X11 下首選圖形🄵覽器，但如果圖形🄵覽器不可用或 X11 顯示不可用，則將使用文字模式🄵覽器。如果使用文字模式🄵覽器，則呼叫程序將會阻塞 (block)，直到使用者退出🄵覽器。

如果環境變數 BROWSER 存在，它會被直譯🄵以*os.pathsep* 分隔的🄵覽器串列，以在平台預設值之前嘗試。當串列部分的值包含字串 `%s` 時，它會被直譯🄵字面🄵覽器命令列，🄵使用引數 URL 替🄵 `%s`；如果值是指向已🄵🄵🄵覽器之一的單個字，則此🄵覽器會被新增到搜尋串列的前面；如果該部分不包含 `%s`，則它僅被直譯🄵要🄵動的🄵覽器的名稱。[1]

在 3.14 版的變更: The BROWSER variable can now also be used to reorder the list of platform defaults. This is particularly useful on macOS where the platform defaults do not refer to command-line tools on PATH.

對於非 Unix 平台，或當 Unix 上有可用的遠端🄵覽器時，控制行程不會等待使用者以完成🄵覽器，而是允許遠端🄵覽器在顯示器上維護自己的視窗。如果遠端🄵覽器在 Unix 上不可用，控制行程將🄵動新的🄵覽器🄵等待。

在 iOS 上，BROWSER 環境變數以及控制自動引發 (autoraise)、🄵覽器設定和新分頁/視窗建立的任何引數都將被忽略。網頁將始終在使用者偏好的🄵覽器中的新分頁中開🄵，🄵將🄵覽器帶到前台。在 iOS 上使用*webbrowser* 模組需要*ctypes* 模組。如果*ctypes* 不可用，則呼叫*open()* 將會失敗。

🄵本 **webbrowser** 可以用作模組的命令列介面。它接受 URL 作🄵引數。它接受以下可選參數：

-n, --new-window

如果可能的話會在新的🄵覽器視窗中開🄵 URL。

[1] 此處命名的無完整路徑可執行檔將在 PATH 環境變數中給定的目🄵中被搜尋。

-t, --new-tab

會在新的🄴覽器分頁 (”tab”) 中開🄵 URL。

這些選項自然是相互排斥的。用法範例如下：

```
python -m webbrowser -t "https://www.python.org"
```

可用性: not WASI, not Android.

以下例外有被定義於該模組：

exception webbrowser.**Error**

當🄴覽器控制項發生錯誤時引發例外。

以下函式有被定義於該模組：

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)

使用預設🄴覽器顯示 *url*。如果 *new* 🄵 0，則盡可能在同一🄴覽器視窗中開🄵 *url*。如果 *new* 🄵 1，則盡可能開🄵一個新的🄴覽器視窗。如果 *new* 🄵 2，則盡可能開🄵一個新的🄴覽器分頁 (”tab”)。如果 *autoraise* 🄵 `True`，則盡可能提升視窗（請注意，無論此變數的設定如何，許多視窗管理器下都會發生這種情🄵）。

如果🄴覽器成功🄴動則回傳 `True`，否則回傳 `False`。

請注意，在某些平台上，嘗試使用此函式開🄴檔案名稱可能能🄴運作🄵🄵動作業系統的關聯程式。然而這既不支援也不可移植。

引發一個附帶引數 `url` 的稽核事件 `webbrowser.open`。

webbrowser.**open_new**(*url*)

盡可能在預設🄴覽器的新視窗中開🄵 *url*，否則在唯一的🄴覽器視窗中開🄵 *url*。

如果🄴覽器成功🄴動則回傳 `True`，否則回傳 `False`。

webbrowser.**open_new_tab**(*url*)

盡可能在預設🄴覽器的新分頁 (”tab”) 中開🄵 *url*，否則相當於*open_new()*。

如果🄴覽器成功🄴動則回傳 `True`，否則回傳 `False`。

webbrowser.**get**(*using=None*)

回傳🄴覽器類型使用 *（以引數 **using* 給定）的控制器物件。如果 *using* 🄵 `None`，則回傳適合呼叫者環境的預設🄴覽器控制器。

webbrowser.**register**(*name*, *constructor*, *instance=None*, *, *preferred=False*)

🄵🄵🄵覽器類型名稱 *（以引數 **name* 給定）。一旦🄵🄵了🄵覽器類型，*get()* 函式就可以回傳該🄵覽器類型的控制器。如果🄵有提供 *instance* 或🄵 `None`，則會在需要時不帶參數地呼叫 *constructor* 來建立實例。如果提供了 *instance*，*constructor* 將永遠不會被呼叫，🄵且可能🄵 `None`。

將 *preferred* 設🄵 `True` 會使該🄴覽器成🄵不帶引數的*get()* 呼叫的偏好結果。否則只有當你計劃設定 `BROWSER` 變數或使用與你宣告的處理程序名稱相符的非空引數呼叫*get()* 時，此入口點才會有用。

在 3.7 版的變更: 新增了 *preferred* 僅限關鍵字參數。

預先定義了多種🄴覽器類型。此表給出了可以傳遞給*get()* 函式的類型名稱以及控制器類🄴的相應實例化方式，這些都定義於此模組中。

類型名懲	類🄵名懲	🄵解
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'epiphany'	Epiphany('epiphany')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'opera'	Opera()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	
'iosbrowser'	IOSBrowser	(4)

🄵解:

(1) "Konqueror" 是 Unix 的 KDE 桌面環境的檔案管理器，只有在 KDE 運作時才有意義。最好能有某種可靠的 KDE 檢測方法；僅有 `KDEDIR` 變數是不🄵的。另請注意，即使在 KDE 2 中使用 **konqueror** 命令時，也會使用"kfm" 名稱 --- 該實作會選擇執行 Konqueror 的最佳策略。

(2) 僅在 Windows 平台上。

(3) 僅在 macOS 上。

(4) 僅在 iOS 上。

在 3.2 版被加入: 新增了 `MacOSXOSAScript` 類🄵🄵於 Mac 上使用，而非使用先前的 `MacOSX` 類🄵。這增加了對開🄵目前未設定🄵作業系統預設之🄵覽器的支援。

在 3.3 版被加入: 新增了對 Chrome/Chromium 的支援。

在 3.12 版的變更: 對多個過時🄵覽器的支援已被🄵除。已🄵除的🄵覽器包括 Grail、Mosaic、Netscape、Galeon、Skipstone、Iceape 和 Firefox 35 及以下版本。

在 3.13 版的變更: 新增了對 iOS 的支援。

以下是一些簡單範例:

```
url = 'https://docs.python.org/'

# 如果🄵覽器視窗已打開，則在新分頁中開🄵 URL。
webbrowser.open_new_tab(url)

# 在新視窗中開🄵 URL，如果可能的話提升視窗。
webbrowser.open_new(url)
```

22.1.1 🄵覽器控制器物件

🄵覽器控制器提供了*name* 屬性和以下三個與模組層級便利函式相同的方法:

`controller.`**`name`**

🄵覽器的系統相依名稱 (system-dependent name)。

controller.**open**(*url*, *new=0*, *autoraise=True*)

使用此控制器處理的🄵覽器顯示 *url*。如果 *new* 🄵 1，則盡可能開🄵一個新的🄵覽器視窗。如果 *new* 🄵 2，則盡可能開🄵一個新的🄵覽器分頁 (”tab”)。

controller.**open_new**(*url*)

盡可能在此控制器處理的🄵覽器的新視窗中開🄵 *url*，否則在唯一的🄵覽器視窗中開🄵 *url*。🄵名🄵*open_new()*。

controller.**open_new_tab**(*url*)

盡可能在此控制器處理的🄵覽器的新分頁 (”tab”) 中開🄵 *url*，否則相當於*open_new()*。

22.2 wsgiref --- WSGI 工具與參考實作

原始碼： Lib/wsgiref

網頁伺服器閘道介面 (WSGI) 是一個標准介面，用於連接網頁伺服器軟體與使用 Python 撰寫的網頁應用程式，擁有一個標准介面使得支援 WSGI 的應用程式可以與多個不同的網頁伺服器運行。

只有網頁伺服器與程式框架的作者需要🄵解 WSGI 設計的每個細節與邊角案例，你🄵不需要🄵了安裝 WSGI 應用程式或是使用現有框架撰寫網頁應用程式而必須理解每個細節。

wsgiref 是 WSGI 規格的參考實作，可用於新增 WSGI 來支援網頁伺服器或框架，它提供操作 WSGI 環境變數以及回應標頭的工具，用於實作 WSGI 伺服器的基本類🄵，提供用於示範 HTTP 伺服器的 WSGI 應用程式、🄵態型🄵檢查、以及驗證 WSGI 伺服器與應用程式是否符合 WSGI 規格的驗證工具 (**PEP 3333**)。

參🄵 wsgi.readthedocs.io 更多 WSGI 相關資訊，以及教學連結與其他資源。

22.2.1 wsgiref.util -- WSGI 環境工具

這個模組提供許多用於處理 WSGI 環境運作的功能。WSGI 環境是一個包含 HTTP 請求變數的字典，如 **PEP 3333** 所述。所有接受 *environ* 的參數的函式都需要提供符合 WSGI 標准的字典；請參🄵 **PEP 3333** 取得詳細規格，以及*WSGIEnvironment* 取得可用於使用型🄵🄵釋的型🄵🄵名。

wsgiref.util.**guess_scheme**(*environ*)

透過檢查 *environ* 字典中的 HTTPS 環境變數，回傳 wsgi.url_scheme 應該是”http” 或”https” 的猜測。回傳值🄵一個字串。

當建立一個包裝 CGI 或類似 FastCGI 的 CGI-like 協定閘道時，此函式非常有用。例如 FastCGI，通常提供這類協定的伺服器在通過 SSL 接收到請求時會包含”1”，”yes”，或”on” 的 HTTPS 變數，因此，如果找到這樣的值，此函式回傳”https”，否則回傳”http”。

wsgiref.util.**request_uri**(*environ*, *include_query=True*)

根據 **PEP 3333** 中”URL Reconstruction” 章節所找到的演算法，回傳完整的請求 URI，可選擇性的包含查詢字串，如果 *include_query* 設🄵 false，查詢字串不會被包含在結果的 URI 中。

wsgiref.util.**application_uri**(*environ*)

類似於*request_uri()*，但忽略 PATH_INFO 和 QUERY_STRING 變數。結果是請求地址的應用程式物件的的基本 URI。

wsgiref.util.**shift_path_info**(*environ*)

將單一名稱從 PATH_INFO 移到 SCRIPT_NAME 🄵回傳該名稱。*environ* 字典會在適當時機被 *modified*；如果你需要保留原始完好無損的 PATH_INFO 或 SCRIPT_NAME 請使用副本。

如果在 PATH_INFO 中🄵有剩余的路徑片段，則回傳 None。

通常，此程式用於處理請求 URI 的每一部分路徑，例如將路徑視🄵一系列的字典鍵此程式會修改傳入的環境，使其適用於呼叫位於目標 URI 的 WSGI 應用程式。例如，如果在 /foo 上有一個 WSGI 應用程式且請求 URI 路徑🄵 /foo/bar/baz，🄵且位於 /foo 的 WSGI 應用程式呼叫*shift_path_info()*，它將接收字串”bar”，而環境將被更新🄵適用於傳遞給位於 /foo/bar 的 WSGI 應用程式。🄵句話🄵，SCRIPT_NAME 將從 /foo 變更🄵 /foo/bar，而 PATH_INFO 將從 /bar/baz 變更🄵 /baz。

當 PATH_INFO 只是一個"/" 時，此程式會回傳一個空字串，🄴在 SCRIPT_NAME 後添加尾部斜號，即使空路徑片段通常是被忽略的，而且 SCRIPT_NAME 通常不會以斜號結尾。這是刻意行🄴，以確保應用程式在使用這個程式進行物件遍歷時可以區分結尾🄴 /x 和結尾🄴 /x/ 的 URIs。

wsgiref.util.**setup_testing_defaults**(*environ*)

🄴測試目的，以簡單的預設值更新 *environ* 。

這個程式新增 WSGI 所需的各種參數，包括 HTTP_HOST、SERVER_NAME、SERVER_PORT、REQUEST_METHOD、SCRIPT_NAME、PATH_INFO，以及所有 **PEP 3333** 定義的 wsgi.* 變數，它只提供預設值，🄴且不會取代現有的這些變數設定。

這個程式目的🄴了讓 WSGI 伺服器和應用程式的單元測試更容易建置🄵擬環境。實際的 WSGI 伺服器或應用程式不應該使用它，因🄴所🄴生的數據是假的！

Example usage (see also *demo_app()* for another example):

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

除了上述的環境功能外，*wsgiref.util* 模組還提供以下各類工具：

wsgiref.util.**is_hop_by_hop**(*header_name*)

如果'header_name' 是根據 **RFC 2616** 所定義的 HTTP/1.1 "Hop-by-Hop" 標頭，則回傳 True。

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

wsgiref.types.FileWrapper 協定的具體實作，用於將類檔案物件轉🄵🄵*iterator*。🄴生的物件是*iterable*。當物件進行🄴代時，將可選的 *blksize* 引數重🄴傳遞給 *filelike* 物件的 read() 方法來獲得將🄵生（yield）的位元組字串。當 read() 回傳一個空位元組字串，代表🄵代已結束且無法回復。

如果 *filelike* 有 close() 方法，則回傳的物件也會具有 close() 方法，🄴在呼叫時叫用 *filelike* 物件的 close() 方法。

用法範例：

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

在 3.11 版的變更: 已移除對 __getitem__() 方法的支援。

22.2.2 `wsgiref.headers` -- WSGI 回應標頭工具

這個模組提供單一類🄴*Headers*，用於使用類似對映的介面方便地操作 WSGI 回應標頭。

class wsgiref.headers.**Headers**(*[headers]*)

建立一個類似對映物件🄴包裝 *headers*，🄴且必須是符合 **PEP 3333** 描述的 name/value 元組的標頭串列。*headers* 的預設值是一個空串列。

Headers 物件支援典型對映操作包括 `__getitem__()`、*get()*、`__setitem__()`、*setdefault()*、`__delitem__()` 以及 `__contains__()`。對於這些方法中的每一個，鍵是標頭名稱（以不區分大小寫方式處理），而值則是與該標頭名稱關聯的第一個值。設定標頭會🄴除該標頭的所有現有值，然後將新值添加到包裝的標頭串列末尾。標頭的現有順序通常保持不變，新標頭會添加到包裝串列的末尾。

不同於字典，當你嘗試取得或🄴除包裝的標頭串列不存在的鍵，*Headers* 物件不會引發例外錯誤。取得不存在的標頭只會回傳 `None`，而🄴除不存在的標頭則不會有任何效果。

Headers 物件還支援 `keys()`、`value()`、和 `items()` 方法。由 `keys()` 和 `items()` 回傳的串列在存在多值標頭時可能會包含相同的鍵。*Headers* 物件的 `len()` 與 `items()` 的長度相同，也與包裝標頭串列的長度相同。實際上，`items()` 方法只是回傳包裝的標頭串列的副本。

對 `Header` 物件呼叫 `bytes()` 會回傳適合作🄵 HTTP 傳輸回應標頭的格式化的位元組字串。每個標頭都與其值一起置於一行上，由冒號與空格分隔。每行以回車（carriage return）和🄵行（line feed）結束，而該位元組字串則以空行結束。

除了對映介面和格式化功能外，*Headers* 物件還具有以下查詢及附加多值標頭的以及附加 MIME 參數標頭的方法：

get_all(*name*)

回傳指定標頭的所有值的串列。

回傳的串列按照它們在在原始的標頭串列出現的順序或是被添加到此實例的順序進行排序，🄵且可能包含重🄵的🄵容。任何被🄵除🄵重新插入的欄位都會被添加到標頭串列的末尾。如果不存在指定名稱的欄位，則回傳空串列。

add_header(*name*, *value*, ***_params*)

添加一個（可能是多值的）標頭，可通過關鍵字引數來指定選擇性的 MIME 參數。

name 是要添加的標頭欄位。關鍵字引數可使於設定標頭欄位的 MIME 參數。每一個參數必須是字串或是 `None`。由於破折號在 Python 識🄵符中是非法的，但是許多 MIME 參數名稱包含破折號，因此參數名稱的底🄵會轉🄵成破折號。如果參數值是字串，則以 `name="value"` 的形式添加到標頭值參數中。如果它是 `None`，則僅添加參數名稱。（這使用於🄵有值的 MIME 參數）使用範例：

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

上述操作將添加看起來像這樣的標頭：

```
Content-Disposition: attachment; filename="bud.gif"
```

在 3.5 版的變更: *headers* 參數是可選的。

22.2.3 `wsgiref.simple_server` -- 一個簡單的 WSGI HTTP 伺服器

這個模組實作一個簡單的的 HTTP 伺服器（基於*http.server*）用於提供 WSGI 應用程式。每個伺服器執行個體在特定的主機與埠提供單一的 WSGI 應用程式。如果你想要在單一主機與埠上提供多個應用程式，你應該建立一個 WSGI 應用程式以剖析 `PATH_INFO` 去選擇🄵每個請求叫用哪個應用程式。（例如，使用來自*wsgiref.util* 的 `shift_path_info()` 函式。）

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

建立一個新的 WSGI 伺服器監聽 *host* 和 *port*，接受 *app* 的連🄵。回傳值是提供 *server_class* 的實例，🄵將使用指定的 *handler_class* 處理請求。*app* 必須是一個 WSGI 應用程式物件，如 **PEP 3333** 所定義。

用法範例：

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

這個函式是一個簡單但完整的 WSGI 應用程式，它回傳一個包含訊息"Hello world!" 和在 *environ* 參數中提供的鍵值對串列的文字頁面。這對於驗證 WSGI 伺服器（例如*wsgiref.simple_server*）是否能正確執行簡單的 WSGI 應用程式非常有用。

The *start_response* callable should follow the *StartResponse* protocol.

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

建立一個*WSGIServer* 實例。*server_address* 應該是一個 (host, port) 元組，而 *RequestHandlerClass* 應該是*http.server.BaseHTTPRequestHandler* 的子類🄵，將用於處理請求。

通常你不需要呼叫這個建構函式（constructor），因🄵*make_server()* 函式可以🄵你處理所有細節。

WSGIServer 是*http.server.HTTPServer* 的子類🄵，因此它的所有方法（例如 serve_forever() 和 handle_request()）都可用。*WSGIServer* 也提供這些特定於 WSGI 的方法：

set_app(*application*)

將可呼叫的 *application* 設定🄵接收請求的 WSGI 應用程式。

get_app()

回傳目前設定應用程式的可呼叫物件。

然而，通常情🄵下你不需要去使用這些額外方法，因🄵*set_app()* 通常會被*make_server()* 呼叫而*get_app()* 主要存在於請求處理程式（handler）實例的好處上。

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

🄵給定的 *request** (即一個 *socket*)、**client_address** (一個 *"(host,port)"* 位元組)、**server* (*WSGIServer* 實例) 建立一個 HTTP 處理程式（handler）。

你不需要直接建立這個類🄵的實例；它們會在需要時由*WSGIServer* 物件自動建立。不過，你可以建立這個類🄵的子類🄵🄵將其作🄵 *handler_class* 提供給*make_server()* 函式。一些可能相關的方法可以在子類🄵中進行覆寫：

get_environ()

唯一個請求回傳一個*WSGIEnvironment* 字典。預設的實作會🄵🄵*WSGIServer* 物件的 base_environ 字典屬性的🄵容以及添加從 HTTP 請求中衍生的各種標頭。每次呼叫這個方法都應該回傳一個包含所有如 **PEP 3333** 所指定的相關 CGI 環境變數的新字典。

get_stderr()

回傳的物件應該被用作 wsgi.errors 串流。預設實作只會回傳 sys.stderr。

handle()

處理 HTTP 請求。預設實作會使用 wsgiref.handler 類🄵來建立處置程式（handler）實例來實作實際 WSGI 應用程式介面。

22.2.4 wsgiref.validate --- WSGI 符合性檢查

當建立新的 WSGI 應用程式物件、框架、伺服器、或是中介軟體 (middleware) 時，使用*wsgiref.validate* 來驗證新程式碼的符合性可能會很有用。這個模組提供一個函式用於建立 WSGI 應用程式物件，🄵用於驗證 WSGI 伺服器或是閘道與 WSGI 應用程式物件之間的通訊，以檢查雙方協定的符合性。

請注意這個工具🄵不保證完全符合 **PEP 3333**；這個模組中的錯誤不一定代表不存在錯誤。但是，如果如果這個模組🄵生錯誤，那🄵幾乎可以確定伺服器或應用程式不是 100% 符合標準。

這個模組基於 Ian Bicking 的"Python Paste" 函式庫的 `paste.lint` 模組。

`wsgiref.validate.`**`validator`**(*application*)

包裝 *application* 🄵回傳一個新的 WSGI 應用程式物件。回傳的應用程式將轉發所有請求給原始的 *application*，🄵檢查 *application* 和呼叫它的伺服器是否符合 WSGI 規範和 **RFC 2616**。

任何在*AssertionError* 中偵測不符合結果都會發起例外；但請注意，如何處理這些錯誤取🄵於伺服器。例如，基於*wsgiref.handlers* 的*wsgiref.simple_server* 以及其他伺服器（未覆蓋錯誤處理方法以執行其他操作的伺服器）將僅輸出一條錯誤訊息，指示發生錯誤，🄵將回溯訊息輸出到 `sys.stderr` 或是其他錯誤串流。

這個包裝器也可以使用*warnings* 模組生成輸出去指示一些可能有疑慮但實際上可能不會被 **PEP 3333** 禁止的行🄵。除非使用 Python 命令列選項或*warnings* API，抑制了這些警告，否則這類警告將被寫入到 `sys.stderr`（*not* `wsgi.errors`，除非它們碰巧是相同的物件）。

用法範例：

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.2.5 `wsgiref.handlers` -- 伺服器 / 閘道基本類🄵

這個模組提供實作 WSGI 伺服器和閘道的基礎處理程式（handler）類🄵。這些基底類🄵處理程式大部分與 WSGI 應用程式通訊的工作，只要它們被提供 CGI-like 環境，以及輸入、輸出和錯誤串流。

`class` `wsgiref.handlers.`**`CGIHandler`**

這是基於 CGI 的呼叫方式🄵透過 `sys.stdin`、`sys.stdout`、`sys.stderr` 和 `os.environ`。當你擁有一個 WSGI 應用程式🄵希望將其作🄵 CGI 🄵本運行時是很有用的。只需叫用 `CGIHandler().run(app)`，其中 `app` 是你希望叫用的 WSGI 應用程式物件。

這個類🄵是*BaseCGIHandler* 的子類🄵將 `wsgi.run_once` 設置🄵 true，`wsgi.multithread` 設置🄵 false，🄵將 `wsgi.multiprocess` 設置🄵 true，🄵且始終使用*sys* 和*os* 來取得所需的 CGI 串流以及環境。

`class` `wsgiref.handlers.`**`IISCGIHandler`**

這是用於在 Microsoft 的 IIS 網頁伺服器上部署時使用的*CGIHandler* 的一個專門替代選擇，無需設置 config 的 allowPathInfo 選項（IIS>=7），或 metabase 的 allowPathInfoForScriptMappings 選項（IIS<7）。

預設情🄵下，IIS 提供的 `PATH_INFO` 會在前面🄵🄵 `SCRIPT_NAME`，對於希望實作路由的 WSGI 應用程式造成問題。這個處理程式（handler）會移除任何這樣的重🄵路徑。

IIS 可以配置去傳遞正確的 `PATH_INFO`，但這會導致 `PATH_TRANSLATED` 是錯誤的問題。幸運的是這個變數很少被使用🄵且不受 WSGI 保證。然而，在 IIS<7 上，這個設置只能在🄵擬主機層級進行，

影響所有其他🄵本的對映，其中許多在暴露 PATH_TRANSLATED 問題時會中斷。由於這個原因幾乎從不會使用修復的 IIS<7（即使是 IIS7 也很少使用它，因🄵它仍然🄵有相應的 UI）。

CGI 程式碼無法知道是否已設置該選項，因此提供了一個獨立的處理程式（handler）類🄵。它的使用方式與*CGIHandler* 相同，即透過呼叫 IISCGIHandler().run(app) 來使用，其中 app 是你希望叫用的 WSGI 應用程式物件。

在 3.2 版被加入.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

類似於*CGIHandler*，但不是使用*sys* 和*os* 模組，而是明確指定 CGI 環境與 I/O 串流。*multithread* 和 *multiprocess* 值用於設置由處理程式（handler）實例運行的任何應用程式的旗標。

這個類🄵是專門🄵除了 HTTP "origin servers" 以外的軟體一起使用的*SimpleHandler* 的子類🄵。如果你正在撰寫一個使用 Status: 標頭來發送 HTTP 狀態的閘道協定實作（例如 CGI、FastCGI、SCGI 等），你可能會想要子類化這個類🄵來替代*SimpleHandler*。

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

類似於*BaseCGIHandler*，但是被設計使用在 HTTP origin 伺服器。如果你正在撰寫 HTTP 伺服器的實作，你可能會想要子類🄵化這個類🄵來替代*BaseCGIHandler*。

這個類🄵是*BaseHandler* 的子類🄵。它透過建構函式去覆寫 __init__()、*get_stdin()*、*get_stderr()*、*add_cgi_vars()*、*_write()*、和*_flush()* 方法來明確提供設置環境與串流。提供的環境與串流被儲存在 stdin、stdout、stderr、和 environ 環境中。

stdout 的*write()* 方法應該完整地寫入每個塊（chunk），像是*io.BufferedIOBase*。

class wsgiref.handlers.**BaseHandler**

這是一個運行 WSGI 應用程式的抽象基底類🄵。每個實例將處理單個 HTTP 請求，🄵管原則上你可以建立一個可重用於多個請求的子類🄵。

BaseHandler 實例只有一個供外部使用的方法：

run(*app*)

運行指定 WSGI 應用程式，*app*。

此方法在運行應用程式的過程中叫用了所有其他*BaseHandler* 的方法，因此這些方法主要存在是🄵了允許自定義整個過程。

以下方法必須在子類🄵中覆寫：

_write(*data*)

緩衝要傳送給用🄵端的位元組 *data*。如果這個方法實際上傳送了數據也是可以的；當底層系統實際具有這種區分時，*BaseHandler* 🄵了更好的效能進而分離寫入和刷新操作。

_flush()

🄵制將緩衝數據傳送到用🄵端。如果這是一個無操作（no-op）的方法（即，如果*_write()* 實際上發送了數據），那🄵是可以的。

get_stdin()

回傳一個與*InputStream* 相容的物件🄵適用於用作目前正在處理請求的 wsgi.input。

get_stderr()

回傳一個與*ErrorStream* 相容的物件🄵適用於用作目前正在處理請求的 wsgi.errors。

add_cgi_vars()

將目前請求的 CGI 變數插入到 environ 屬性中。

以下是你可能希望覆寫的其他方法和屬性。這個列表只是一個摘要，然而，不包括可以被覆寫的每個方法。在嘗試建立自定義的*BaseHandler* 子類🄵之前，你應該參考文件🄵明和原始碼以獲得更多資訊。

用於自定義 WSGI 環境的屬性和方法：

wsgi_multithread

用於 wsgi.multithread 環境變數的值。在*BaseHandler* 中預設🄵 true，但在其他子類🄵中可能有不同的預設值（或由建構函式設置）。

wsgi_multiprocess

用於 wsgi.multiprocess 環境變數的值。在*BaseHandler* 中預設🄵 true，但在其他子類🄵中可能有不同的預設值（或由建構函式設置）。

wsgi_run_once

用於 wsgi.run_once 環境變數的值。在*BaseHandler* 中預設🄵 false，但*CGIHandler* 預設將其設置🄵 true。

os_environ

預設環境變數包含在每一個請求的 WSGI 環境中。預設情🄵下，這是在載入*wsgiref.handlers* 時的 os.environ 副本，但子類🄵可以在類🄵或實例層級建立自己的副本。注意字典應該被視🄵唯讀，因🄵預設值在多個類🄵與實例中共享。

server_software

如果設置*origin_server* 屬性，則此屬性的值將用於設置預設的 SERVER_SOFTWARE WSGI 環境變數，🄵且還將用於設置 HTTP 回應中的預設 Server: 標頭。對於不是 HTTP origin 伺服器的處置程式（例如*BaseCGIHandler* 和*CGIHandler*），此屬性將被忽略。

在 3.3 版的變更: 將術語”Python” 替🄵🄵特定實作的術語，如”CPython”、”Jython” 等。

get_scheme()

回傳用於目前請求的 URL scheme。預設的實作使用*wsgiref.util* 中的 guess_scheme() 函式去猜測 scheme 是”http” 或是”https”，基於目前請求的 environ 變數。

setup_environ()

將 environ 屬性設置🄵完全填充的 WSGI 環境。預設的實作使用上述所有方法和屬性，以及*get_stdin()*、*get_stderr()* 和*add_cgi_vars()* 方法以及*wsgi_file_wrapper* 屬性。如果不呈現它也會插入一個 SERVER_SOFTWARE 關鍵字，只要*origin_server* 屬性是一個 true 值🄵且*server_software* 屬性被設置。

用於自定義例外處理的屬性和方法：

log_exception(*exc_info*)

將 *exc_info* 元組記🄵到伺服器日🄵中。*exc_info* 是一個 (type, value, traceback) 元組。預設實作只是將追🄵資訊寫入到請求的 wsgi.errors 串流中🄵刷新它。子類🄵可以覆蓋此方法以更改格式或重新定向輸出，將追🄵資訊發送給管理員，或執行其他被認🄵合適的操作。

traceback_limit

預設的*log_exception()* 方法追🄵輸出中包含的最大幀數。如果🄵 None，則包含所有幀。

error_output(*environ*, *start_response*)

這個方法是一個🄵使用者去🄵生錯誤頁面的 WSGI 應用程式。只有在標頭傳送給用🄵端前如果發生錯誤才會被叫用。

此方法使用 sys.exception() 存取目前的錯誤，當呼叫它（如 **PEP 3333** 的”Error Handling” 部分所描述）時應該傳遞資訊給 *start_response*。特🄵是 *start_response* 可呼叫物件應該遵循*StartResponse* 協定。

預設的實作只是使用*error_status*、*error_headers* 和*error_body* 屬性🄵生輸出頁面。子類🄵可以覆蓋此方法以生成更動態的錯誤輸出。

然而，從安全的角度🄵不建議向任何普通使用者顯示診斷資訊；理想情🄵下，你應該需要🄵取特殊措施才能🄵用診斷輸出，這就是預設實作不包括任何診斷資訊的原因。

error_status

用於錯誤回應的 HTTP 狀態。這應該是一個按照 **PEP 3333** 定義的狀態字串；預設🄵 500 狀態碼和訊息。

error_headers

用於錯誤回應的 HTTP 標頭。這應該是一個 WSGI 回應標頭的串列（`(name, value)` 元組），如 **PEP 3333** 中所描述。預設串列只設置🄵容種類🄵 `text/plain`。

error_body

錯誤回應的主體。這應該是一個 HTTP 回應🄵容的位元組字串。預設🄵純文字"A server error occurred. Please contact the administrator."

用於 **PEP 3333** 中的"Optional Platform-Specific File Handling" 功能的方法和屬性：

wsgi_file_wrapper

一個 `wsgi.file_wrapper` 工廠函式（factory），與*wsgiref.types.FileWrapper* 相容，或者🄵 `None`。這個屬性的預設值是*wsgiref.util.FileWrapper* 類🄵。

sendfile()

覆蓋以實作特定平台的檔案傳輸。只有當應用程式的回傳值是由*wsgi_file_wrapper* 屬性指定的類🄵實例時才會呼叫此方法。如果它能🄵成功傳輸檔案應該回傳一個 true 值，以便不執行預設的傳輸程式碼。該方法的預設實作只回傳一個 false 值。

其他方法和屬性：

origin_server

這個屬性應該被設置🄵 true 值，如果處理程式（handler）的*_write()* 和*_flush()* 被用於直接與用🄵端通訊，而不是透過 CGI-like 的閘道協定希望 HTTP 狀態在特殊的 `Status:` 標頭中。

這個屬性在*BaseCGIHandler* 預設值🄵 true，但是在*BaseCGIHandler* 和*CGIHandler* 🄵 false。

http_version

如果*origin_server* 🄵 true，則此字串屬性用於設定傳送給用🄵端的回應的 HTTP 版本。預設🄵 `"1.0"`。

wsgiref.handlers.**read_environ**()

從 `os.environ` 轉碼 CGI 變數到 **PEP 3333** 中的"bytes in unicode" 字串，🄵回傳一個新字典。這個函式被*CGIHandler* 和*IISCGIHandler* 使用來直接替代 `os.environ`，在所有平台和使用 Python 3 的網頁伺服器上不一定符合 WSGI 標准，具體來🄵，在 OS 的實際環境是 Unicode（例如 Windows）的情🄵下，或者在環境是位元組的情🄵下，但 Python 用於解碼它的系統編碼不是 ISO-8859-1 （例如使用 UTF-8 的 Unix 系統）。

如果你自己正在實作 CGI-based 處理程式（handler），你可能想要使用這個函式來替🄵單純直接從 `os.environ` 中🄵🄵值。

在 3.2 版被加入.

22.2.6 wsgiref.types -- 用於🄵態型🄵檢查的 WSGI 型🄵

這個模組提供在 **PEP 3333** 中所描述的各種用於🄵態型🄵檢查的型🄵。

在 3.11 版被加入.

class wsgiref.types.**StartResponse**

一個描述 **start_response()** 可呼叫物件的*typing.Protocol* (**PEP 3333**)。

wsgiref.types.**WSGIEnvironment**

一個描述 WSGI 環境字典的型🄵🄵名。

wsgiref.types.**WSGIApplication**

一個描述 WSGI 應用程式可呼叫物件的型🄵🄵名。

class wsgiref.types.**InputStream**

一個描述 **WSGI 輸入串流**的*typing.Protocol*。

class wsgiref.types.**ErrorStream**

一個描述 **WSGI 錯誤串流**的*typing.Protocol*。

class wsgiref.types.**FileWrapper**

一個描述**檔案包裝器**的*typing.Protocol*。請參🄵*wsgiref.util.FileWrapper* 來🄵解此協定的具體實作。

22.2.7 范例

This is a working "Hello World" WSGI application, where the *start_response* callable should follow the *StartResponse* protocol:

```
"""
Every WSGI application must have an application object - a callable
object that accepts two arguments. For that purpose, we're going to
use a function (note that you're not limited to a function, you can
use a class for example). The first argument passed to the function
is a dictionary containing CGI-style environment variables and the
second variable is the callable object.
"""
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

提供目前目🄵的 WSGI 應用程式範例，🄵接受命令列上的可選目🄵和埠號（預設：8000）：

```
"""
Small wsgiref based web server. Takes a path to serve from and an
optional port number (defaults to 8000), then tries to serve files.
MIME types are guessed from the file names, 404 errors are raised
if the file is not found.
"""
import mimetypes
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # Get the file name and MIME type
    fn = os.path.join(path, environ["PATH_INFO"][1:])
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # Return 200 OK if file exists, otherwise 404 Not Found
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type", "text/plain")])
        return [b"not found"]
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == "__main__":
    # Get the path and port from command-line arguments
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # Make and start the server until control-c
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

22.3 `urllib` --- URL 處理模組

原始碼： Lib/urllib/

`urllib` 是一個🄵集了許多處理 URLs 的 module（模組）的 package（套件）：

- *`urllib.request`* 用來開🄵和讀取 URLs
- *`urllib.error`* 包含了*`urllib.request`* 所引發的例外
- *`urllib.parse`* 用來剖析 URLs
- *`urllib.robotparser`* 用來剖析 `robots.txt` 檔案

22.4 `urllib.request` --- 用來開🄵 URLs 的可擴充函式庫

原始碼： Lib/urllib/request.py

`urllib.request` module（模組）定義了一些函式與 class（類🄵）用以開🄵 URLs（大部分是 HTTP），🄵處理各式🄵雜情🄵如：basic 驗證與 digest 驗證、重新導向、cookies。

也參考

有關於更高階的 HTTP 用🄵端介面，推薦使用 Requests 套件。

警告

On macOS it is unsafe to use this module in programs using *`os.fork()`* because the *`getproxies()`* implementation for macOS uses a higher-level system API. Set the environment variable `no_proxy` to `*` to avoid this problem (e.g. `os.environ["no_proxy"] = "*"`).

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

`urllib.request` module 定義下列函式：

urllib.request.**urlopen**(*url*, *data=None*, [*timeout*,]*, *context=None*)

打開 *url*，其值可以是一個包含有效且適當編碼 URL 的字串或是一個*Request* 物件。

data 必須是一個包含傳送給伺服器額外資料的物件，若不需要傳送額外資料則指定🄵 None。更多細節請見*Request*。

urllib.request module 使用 HTTP/1.1 🄵包含 Connection:close header（標頭）在其 HTTP 請求中。

透過選擇性參數 *timeout* 來指定 blocking operations（阻塞性操作，如：嘗試連接）的 timeout（超時時間），以秒🄵單位。若🄵有指定值，則會使用全域預設超時時間設定。實際上，此參數僅作用於 HTTP、HTTPS 以及 FTP 的連接。

若 *context* 有被指定時，它必須是一個*ssl.SSLContext* 的實例🄵描述著各種 SSL 選項。更多細節請見*HTTPSConnection*。

這個函式總是回傳一個可作🄵*context manager* 使用的物件，🄵有著特性 (property) *url*、*headers* 與 *status*。欲知更多這些特性細節請參見*urllib.response.addinfourl*。

對於 HTTP 與 HTTPS 的 URLs，這個函式回傳一個稍有不同的*http.client.HTTPResponse* 物件。除了上述提到的三個方法外，另有 msg 屬性🄵有著與*reason* 相同的資訊 --- 由伺服器回傳的原因🄵述 (reason phrase)，而不是在*HTTPResponse* 文件中提到的回應 headers。

對於 FTP、檔案及資料的 URLs，這個函式會回傳一個*urllib.response.addinfourl* 物件。

當遇到協定上的錯誤時會引發*URLError*。

請注意若🄵有 handler 處理請求時，None 值將會被回傳。（即使有預設的全域類🄵*OpenerDirector* 使用*UnknownHandler* 來確保這種情🄵不會發生）

另外，若有偵測到代理服務的設定（例如當 *_proxy 環境變數像是：:envvar:!http_proxy‘ 有被設置時），*ProxyHandler* 會被預設使用以確保請求有透過代理服務來處理。

Python 2.6 或更早版本的遺留函式 urllib.urlopen 已經不再被維護；新函式*urllib.request.urlopen()* 對應到舊函式 urllib2.urlopen。有關代理服務的處理，以往是透過傳遞 dictionary（字典）參數給 urllib.urlopen 來取得的，現在則可以透過*ProxyHandler* 物件來取得。

預設的 opener 會觸發一個*auditing event* urllib.Request 與其從請求物件中所獲得的引數 fullurl、data、headers、method。

在 3.2 版的變更: 新增 *cafile* 與 *capath*。

HTTPS 🄵擬主機 (virtual hosts) 現已支援，只要*ssl.HAS_SNI* 的值🄵 true。

data 可以是一個可🄵代物件。

在 3.3 版的變更: *cadefault* 被新增。

在 3.4.3 版的變更: *context* 被新增。

在 3.10 版的變更: 當 *context* 🄵有被指定時，HTTPS 連🄵現在會傳送一個帶有協定指示器 http/1.1 的 ALPN 擴充 (extension)。自訂的 *context* 應該利用*set_alpn_protocols()* 來自行設定 ALPN 協定。

在 3.13 版的變更: Remove *cafile*, *capath* and *cadefault* parameters: use the *context* parameter instead.

urllib.request.**install_opener**(*opener*)

安裝一個*OpenerDirector* 實例作🄵預設的全域 opener。僅在當你想要讓 urlopen 使用該 opener 時安裝一個 opener，否則的話應直接呼叫*OpenerDirector.open()* 而非*urlopen()*。程式碼不會檢查 class 是否真的🄵*OpenerDirector*，而是任何具有正確介面的 class 都能適用。

urllib.request.**build_opener**([*handler*, ...])

回傳一個*OpenerDirector* 實例，以給定的順序把 handlers 串接起來。*handler*s 可以是*BaseHandler* 的實例，亦或是*BaseHandler* 的 subclasses（這個情🄵下必須有不帶參數的建構函式能🄵被呼叫）。以下 classes 的實例順位會在 *handler*s 之前，除非 *handler*s 已經包含它們，是它們的實例，或是它們的 subclasses：*ProxyHandler*（如果代理服務設定被偵測到）、*UnknownHandler*、*HTTPHandler*、*HTTPDefaultErrorHandler*、*HTTPRedirectHandler*、*FTPHandler*、*FileHandler*、*HTTPErrorProcessor*。

如果 Python 安裝時已帶有 SSL 支援（如果*ssl* module 能🄴被 import），則*HTTPSHandler* 也在上述 class 之中。

一個*BaseHandler* 的 subclass 可能透過改變其 `handler_order` 屬性來調整它在 handlers list 中的位置。

`urllib.request.`**`pathname2url`**(*path*, *, *add_scheme=False*)

Convert the given local path to a `file:` URL. This function uses *quote()* function to encode the path.

If *add_scheme* is false (the default), the return value omits the `file:` scheme prefix. Set *add_scheme* to true to return a complete URL.

This example shows the function being used on Windows:

```
>>> from urllib.request import pathname2url
>>> path = 'C:\\Program Files'
>>> pathname2url(path, add_scheme=True)
'file:///C:/Program%20Files'
```

在 3.14 版的變更: Windows drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an *OSError* exception to be raised on Windows.

在 3.14 版的變更: Paths beginning with a slash are converted to URLs with authority sections. For example, the path `/etc/hosts` is converted to the URL `///etc/hosts`.

在 3.14 版的變更: The *add_scheme* parameter was added.

`urllib.request.`**`url2pathname`**(*url*, *, *require_scheme=False*, *resolve_host=False*)

Convert the given `file:` URL to a local path. This function uses *unquote()* to decode the URL.

If *require_scheme* is false (the default), the given value should omit a `file:` scheme prefix. If *require_scheme* is set to true, the given value should include the prefix; a *URLError* is raised if it doesn't.

The URL authority is discarded if it is empty, `localhost`, or the local hostname. Otherwise, if *resolve_host* is set to true, the authority is resolved using *socket.gethostbyname()* and discarded if it matches a local IP address (as per **RFC 8089 §3**). If the authority is still unhandled, then on Windows a UNC path is returned, and on other platforms a *URLError* is raised.

This example shows the function being used on Windows:

```
>>> from urllib.request import url2pathname
>>> url = 'file:///C:/Program%20Files'
>>> url2pathname(url, require_scheme=True)
'C:\\Program Files'
```

在 3.14 版的變更: Windows drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an *OSError* exception to be raised on Windows.

在 3.14 版的變更: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a *URLError* is raised.

在 3.14 版的變更: The URL query and fragment components are discarded if present.

在 3.14 版的變更: The *require_scheme* and *resolve_host* parameters were added.

`urllib.request.`**`getproxies`**()

這個輔助函式 (helper function) 回傳一個代理伺服器 URL mappings（對映）的 dictionary。在所有的作業系統中，它首先掃描環境中有著 `<scheme>_proxy` 名稱的變數（忽略大小寫的），如果找不到的話就會在 macOS 中的系統設定 (System Configuration) 或是 Windows 系統中的 Windows Systems Registry 尋找代理服務設定。如果大小寫的環境變數同時存在且值有不同，小寫的環境變數會被選用。

> **備🄵**
>
> 如果環境變數 REQUEST_METHOD 有被設置（通常這代表著你的 script 是運行在一個共用閘道介面 (CGI) 環境中），那🄵環境變數 HTTP_PROXY （大寫的 _PROXY）將被忽略。這是因🄵變數可以透過使用"Proxy:" HTTP header 被注入。如果需要在共用閘道介面環境中使用 HTTP 代理服務，可以明確使用 ProxyHandler，亦或是確認變數名稱是小寫的（或至少 _proxy 後綴是小寫的）。

提供了以下的 classes：

class urllib.request.**Request**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

這個 class 是一個 URL 請求的抽象 class。

url 是一個包含有效且適當編碼的 URL 字串。

data 必須是一個包含要送到伺服器的附加資料的物件，若不需帶附加資料則其值應🄵 None。目前 HTTP 請求是唯一有使用 *data* 參數的，其支援的物件型🄵包含位元組、類檔案物件 (file-like objects)、以及可🄵代的類位元組串物件 (bytes-like objects)。如果🄵有提供 Content-Length 及 Transfer-Encoding headers 欄位，*HTTPHandler* 將會根據 *data* 的型🄵設置這些 header。Content-Length 會被用來傳送位元組串物件，而 **RFC 7230** 章節 3.3.1 所定義的 Transfer-Encoding: chunked 則會被用來傳送檔案或是其它可🄵代物件 (iterables)。

對於一個 HTTP POST 請求方法，*data* 應🄵一個標准 *application/x-www-form-urlencoded* 格式的 buffer。*urllib.parse.urlencode()* 方法接受一個 mapping 或是 sequence（序列）的 2-tuples，🄵回傳一個對應格式的 ASCII 字串。在被作🄵 *data* 參數前它應該被編碼成位元組串。

headers 必須是一個 dictionary，🄵會被視🄵如同每對 key 和 value 作🄵引數來呼叫*add_header()*。經常用於「🄵裝」User-Agent header 的值，這個 header 是用來讓一個🄵覽器向伺服器表明自己的身分 --- 有些 HTTP 伺服器僅允許來自普通🄵覽器的請求，而不接受來自程式🄵本的請求。例如，Mozilla Firefox 會將 header 的值設🄵 "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"，而*urllib* 的值則是 "Python-urllib/2.6"（在 Python 2.6 上）。所有 header 的鍵都會以 camel case（駝峰式大小寫）來傳送。

當有給定 *data* 引數時，一個適當的 Content-Type header 應該被設置。如果這個 header 🄵有被提供且 *data* 也不🄵 None 時，預設值 Content-Type: application/x-www-form-urlencoded 會被新增至請求中。

接下來的兩個引數的介紹提供給那些有興趣正確處理第三方 HTTP cookies 的使用者：

origin_req_host 應🄵原始傳輸互動的請求主機 (request-host)，如同在 **RFC 2965** 中的定義。預設值🄵 http.cookiejar.request_host(self)。這是使用者發起的原始請求的主機名稱或是 IP 位址。例如當請求是要求一個 HTML 文件中的一個影像，則這個屬性應🄵請求包含影像頁面的請求主機。

unverifiable 應該標示一個請求是否是無法驗證的，如同在 **RFC 2965** 中的定義。其預設值🄵 False。一個無法驗證的請求是指使用者🄵有機會去批准請求的 URL，例如一個對於 HTML 文件中的影像所做的請求，而使用者🄵有機會去批准是否能自動🄵取影像，則這個值應該🄵 true。

method 應🄵一個標示 HTTP 請求方法的字串（例如：'HEAD'）。如果有提供值，則會被存在*method* 屬性中且被*get_method()* 所使用。當 *data* 是 None 時，其預設值🄵 'GET'，否則預設值🄵 'POST'。Subclasses 可以透過設置其*method* 屬性來設定不一樣的預設請求方法。

> **備🄵**
>
> 如果資料物件無法重🄵提供其🄵容（例如一個檔案或是只能🄵生一次🄵容的可🄵代物件）且請求因🄵 HTTP 重導向 (redirects) 或是 HTTP 驗證 (authentication) 而被重新嘗試傳送，則該請求不會正常運作。*data* 會接在 headers 之後被送至 HTTP 伺服器。此函式庫🄵有支援 100-continue expectation。

在 3.3 版的變更: 新增*Request.method* 引數到 Request class。

在 3.4 版的變更: 能🄵在 class 中設置預設的*Request.method*。

在 3.6 版的變更: 如果 Content-Length 尚未被提供且 *data* 既不是 None 也不是一個位元組串物件，則不會觸發錯誤，🄵 fall back（後備）使用分塊傳輸編碼 (chunked transfer encoding)。

class urllib.request.**OpenerDirector**

The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.**BaseHandler**

This is the base class for all registered handlers --- and handles only the simple mechanics of registration.

class urllib.request.**HTTPDefaultErrorHandler**

A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class urllib.request.**HTTPRedirectHandler**

A class to handle redirections.

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)

A class to handle HTTP Cookies.

class urllib.request.**ProxyHandler**(*proxies=None*)

Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables <protocol>_proxy. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a macOS environment proxy information is retrieved from the System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The no_proxy environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with :port appended, for example cern.ch,ncsa.uiuc.edu,some.host:8080.

> **備🄵**
>
> HTTP_PROXY will be ignored if a variable REQUEST_METHOD is set; see the documentation on *getproxies()*.

class urllib.request.**HTTPPasswordMgr**

Keep a database of (realm, uri) -> (user, password) mappings.

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**

Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of uri -> is_authenticated mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a 401 response first.

在 3.5 版被加入.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported. If *passwd_mgr* also provides is_authenticated and update_authenticated methods (see *HTTPPasswordMgrWithPriorAuth 物件*), then the handler will use the is_authenticated result for a given URI to determine whether or not to send authentication credentials with the request. If is_authenticated returns True for the URI, credentials are sent. If is_authenticated is False, credentials are not sent, and then if a 401 response is received the request is re-sent with the authentication credentials. If authentication succeeds, update_authenticated

is called to set `is_authenticated` `True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

在 3.5 版被加入: 新增 `is_authenticated` 的支援。

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported.

在 3.14 版的變更: Added support for HTTP digest authentication algorithm `SHA-256`.

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

在 3.3 版的變更: Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr 物件* for information on the interface that must be supported.

class urllib.request.**HTTPHandler**

A class to handle opening of HTTP URLs.

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)

A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

在 3.2 版的變更: 新增 *context* 與 *check_hostname*。

class urllib.request.**FileHandler**

Open local files.

class urllib.request.**DataHandler**

Open data URLs.

在 3.4 版被加入.

class urllib.request.**FTPHandler**

打開 FTP URLs。

class urllib.request.**CacheFTPHandler**

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.**UnknownHandler**

A catch-all class to handle unknown URLs.

class urllib.request.**HTTPErrorProcessor**

Process HTTP error responses.

22.4.1 Request 物件

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.**full_url**

The original URL passed to the constructor.

在 3.4 版的變更.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.**type**

The URI scheme.

Request.**host**

The URI authority, typically a host, but may also contain a port separated by a colon.

Request.**origin_req_host**

The original host for the request, without port.

Request.**selector**

The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.**data**

The entity body for the request, or None if not specified.

在 3.4 版的變更: Changing value of *Request.data* now deletes "Content-Length" header if it was previously set or calculated.

Request.**unverifiable**

boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.**method**

The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

在 3.3 版被加入.

在 3.4 版的變更: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.**get_method**()

Return a string indicating the HTTP request method. If *Request.method* is not None, return its value, otherwise return 'GET' if *Request.data* is None, or 'POST' if it's not. This is only meaningful for HTTP requests.

在 3.3 版的變更: get_method now looks at the value of *Request.method*.

Request.**add_header**(*key*, *val*)

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

Request.**add_unredirected_header**(*key*, *header*)

Add a header that will not be added to a redirected request.

Request.**has_header**(*header*)

Return whether the instance has the named header (checks both regular and unredirected).

Request.**remove_header**(*header*)

Remove named header from the request instance (both from regular and unredirected headers).

在 3.4 版被加入.

Request.**get_full_url**()

Return the URL given in the constructor.

在 3.4 版的變更.

回傳*Request.full_url*

Request.**set_proxy**(*host*, *type*)

Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.**get_header**(*header_name*, *default=None*)

Return the value of the given header. If the header is not present, return the default value.

Request.**header_items**()

Return a list of tuples (header_name, header_value) of the Request headers.

在 3.4 版的變更: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

22.4.2 OpenerDirector 物件

OpenerDirector 物件有以下的方法:

OpenerDirector.**add_handler**(*handler*)

handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example `http_response()` would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example `http_error_404()` would handle HTTP 404 errors.

- `<protocol>_open()` --- signal that the handler knows how to open *protocol* URLs.

  更多資訊請見*BaseHandler.<protocol>_open()*。

- `http_error_<type>()` --- signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

  更多資訊請見*BaseHandler.http_error_<nnn>()*。

- `<protocol>_error()` --- signal that the handler knows how to handle errors from (non-`http`) *protocol*.

- `<protocol>_request()` --- signal that the handler knows how to pre-process *protocol* requests.

  更多資訊請見*BaseHandler.<protocol>_request()*。

- `<protocol>_response()` --- signal that the handler knows how to post-process *protocol* responses.

  更多資訊請見*BaseHandler.<protocol>_response()*。

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

OpenerDirector.**error**(*proto*, **args*)

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_<type>()` methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `<protocol>_request()` has that method called to pre-process the request.
2. Handlers with a method named like `<protocol>_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named *default_open()*. If all such methods return *None*, the algorithm is repeated for methods named like `<protocol>_open()`. If all such methods return *None*, the algorithm is repeated for methods named *unknown_open()*.

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.
3. Every handler with a method named like `<protocol>_response()` has that method called to post-process the response.

22.4.3 BaseHandler 物件

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler.**add_parent**(*director*)

Add a director as parent.

BaseHandler.**close**()

Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

> **備🄵**
>
> The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

BaseHandler.**parent**

A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

BaseHandler.**default_open**(*req*)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* method of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to *URLError*).

This method will be called before any protocol-specific open method.

BaseHandler.<protocol>_open(req)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**unknown_open**(*req*)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent* *OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)

This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

OpenerDirector 會以五個位置引數呼叫此方法：

1. 一個*Request* 物件
2. 一個類檔案物件，包含 HTTP 錯誤主體，
3. HTTP 錯誤的三位數代號字串，
4. HTTP 錯誤代號的使用者可見解釋字串，以及
5. HTTP 錯誤的標頭，🄵一個對映物件。

Return values and exceptions raised should be the same as those of *urlopen()*.

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)

nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for *http_error_default()*.

BaseHandler.<protocol>_request(req)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

BaseHandler.<protocol>_response(req, response)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

22.4.4 HTTPRedirectHandler 物件

備🄵

Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

An *HTTPError* exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise *HTTPError* if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

> **備🄵**
>
> The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

Redirect to the `Location:` or `URI:` URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'found' response.

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'see other' response.

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'temporary redirect' response. It does not allow changing the request method from `POST` to `GET`.

HTTPRedirectHandler.**http_error_308**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as *http_error_301()*, but called for the 'permanent redirect' response. It does not allow changing the request method from `POST` to `GET`.

在 3.11 版被加入.

22.4.5 HTTPCookieProcessor 物件

HTTPCookieProcessor 實例有一個屬性:

HTTPCookieProcessor.**cookiejar**

存放 cookies 的*http.cookiejar.CookieJar*。

22.4.6 ProxyHandler 物件

ProxyHandler.<protocol>_open(request)

The *ProxyHandler* will have a method `<protocol>_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

22.4.7 HTTPPasswordMgr 物件

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

22.4.8 HTTPPasswordMgrWithPriorAuth 物件

This password manager extends *HTTPPasswordMgrWithDefaultRealm* to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd* are as for *HTTPPasswordMgr.add_password()*. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

Same as for *HTTPPasswordMgrWithDefaultRealm* objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

Returns the current state of the `is_authenticated` flag for the given URI.

22.4.9 AbstractBasicAuthHandler 物件

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

22.4.10 HTTPBasicAuthHandler 物件

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.11 ProxyBasicAuthHandler 物件

ProxyBasicAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.12 AbstractDigestAuthHandler 物件

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

22.4.13 HTTPDigestAuthHandler 物件

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.14 ProxyDigestAuthHandler 物件

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.15 HTTPHandler 物件

`HTTPHandler.`**`http_open`**(*req*)

Send an HTTP request, which can be either GET or POST, depending on `req.data`.

22.4.16 HTTPSHandler 物件

`HTTPSHandler.`**`https_open`**(*req*)

Send an HTTPS request, which can be either GET or POST, depending on `req.data`.

22.4.17 FileHandler 物件

`FileHandler.`**`file_open`**(*req*)

Open the file locally, if there is no host name, or the host name is `'localhost'`.

在 3.2 版的變更: This method is applicable only for local hostnames. When a remote hostname is given, a *URLError* is raised.

22.4.18 DataHandler 物件

`DataHandler.`**`data_open`**(*req*)

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise a *ValueError* in that case.

22.4.19 FTPHandler 物件

`FTPHandler.`**`ftp_open`**(*req*)

Open the FTP file indicated by *req*. The login is always done with empty username and password.

22.4.20 CacheFTPHandler 物件

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods:

`CacheFTPHandler.`**`setTimeout`**(*t*)

Set timeout of connections to *t* seconds.

`CacheFTPHandler.`**`setMaxConns`**(*m*)

Set maximum number of cached connections to *m*.

22.4.21 UnknownHandler 物件

`UnknownHandler.`**`unknown_open`**()

Raise a *URLError* exception.

22.4.22 HTTPErrorProcessor 物件

`HTTPErrorProcessor.`**`http_response`**(*request*, *response*)

Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `http_error_<type>()` handler methods, via *`OpenerDirector.error()`*. Eventually, *`HTTPDefaultErrorHandler`* will raise an *`HTTPError`* if no other handler handles the error.

`HTTPErrorProcessor.`**`https_response`**(*request*, *response*)

Process HTTPS error responses.

The behavior is same as *`http_response()`*.

22.4.23 范例

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it:

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <!
↪[endif]-->\n<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]--
↪>\n<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following HTML spec document, https://html.spec.whatwg.org/#charset, lists the various ways in which an HTML or an XML document could have specified its encoding information.

For additional information, see the W3C document: https://www.w3.org/International/questions/qa-html-encoding-declarations.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object:

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!-
```

It is also possible to achieve the same result without using the *context manager* approach:

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> try:
...     print(f.read(100).decode('utf-8'))
... finally:
...     f.close()
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *Request*:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

使用基本 HTTP 認證:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
with urllib.request.urlopen('http://www.example.com/login.html') as f:
    print(f.read().decode('utf-8'))
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# 這次我們直接使用它而不安裝 OpenerDirector:
with opener.open('http://www.example.com/login.html') as f:
   print(f.read().decode('utf-8'))
```

添加 HTTP 標頭:

使用*Request* 建構函式的 *headers* 引數，或:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
```

(繼續下一頁)

(繼續上一頁)

```
req.add_header('Referer', 'http://www.python.org/')
# 自訂預設的使用者代理標頭值:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
with urllib.request.urlopen(req) as f:
    print(f.read().decode('utf-8'))
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
with opener.open('http://www.example.com/') as f:
   print(f.read().decode('utf-8'))
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler(proxies))
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler({}))
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.4.24 遺留介面

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

urllib.request.**urlretrieve**(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *urlopen()* returned (for a remote object). Exceptions are the same as for *urlopen()*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise *ContentTooShortError* when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

urllib.request.**urlcleanup**()

Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

22.4.25 `urllib.request` 限制

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  在 3.4 版的變更: Added support for data URLs.

- The caching feature of *urlretrieve()* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *html.parser* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL

ends in a /, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing / has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module.

22.5 urllib.response --- Response classes used by urllib

The *urllib.response* module defines functions and classes which define a minimal file-like interface, including read() and readline(). Functions defined by this module are used internally by the *urllib.request* module. The typical response object is a *urllib.response.addinfourl* instance:

class urllib.response.**addinfourl**

url

URL of the resource retrieved, commonly used to determine if a redirect was followed.

headers

Returns the headers of the response in the form of an *EmailMessage* instance.

status

在 3.9 版被加入.

伺服器回傳的狀態碼。

geturl()

在 3.9 版之後被🄵用: 已被*url* 取代而🄵用。

info()

在 3.9 版之後被🄵用: 已被*headers* 取代而🄵用。

code

在 3.9 版之後被🄵用: 已被*status* 取代而🄵用。

getcode()

在 3.9 版之後被🄵用: 已被*status* 取代而🄵用。

22.6 urllib.parse --- 將 URL 剖析成元件

原始碼： Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: file, ftp, gopher, hdl, http, https, imap, itms-services, mailto, mms, news, nntp, prospero, rsync, rtsp, rtsps, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais, ws, wss.

CPython 實作細節： The inclusion of the itms-services URL scheme can prevent an app from passing Apple's App Store review process for the macOS and iOS App Stores. Handling for the itms-services scheme is always removed on iOS; on macOS, it *may* be removed if CPython has been built with the --with-app-store-compliance option.

The *urllib.parse* module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

This module's functions use the deprecated term `netloc` (or `net_loc`), which was introduced in **RFC 1808**. However, this term has been obsoleted by **RFC 3986**, which introduced the term `authority` as its replacement. The use of `netloc` is continued for backward compatibility.

22.6.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

`urllib.parse.`**`urlparse`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up into smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is a *named tuple*, which means that its items can be accessed by index or as named attributes, which are:

屬性	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace()`. The `_replace()` method will return a new ParseResult object replacing specified fields with new values.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

警告

urlparse() does not perform validation. See *URL parsing security* for details.

在 3.2 版的變更: 新增剖析 IPv6 URL 的能力。

在 3.3 版的變更: The fragment is now parsed for all URL schemes (unless *allow_fragments* is false), in accordance with **RFC 3986**. Previously, an allowlist of schemes that support fragments existed.

在 3.6 版的變更: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

在 3.8 版的變更: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

`urllib.parse.`**`parse_qs`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

在 3.2 版的變更: 新增 *encoding* 和 *errors* 參數。

在 3.8 版的變更: 新增 *max_num_fields* 參數。

在 3.10 版的變更: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

在 3.14 版之後被🄴用: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

`urllib.parse.`**`parse_qsl`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

在 3.2 版的變更: 新增 *encoding* 和 *errors* 參數。

在 3.8 版的變更: 新增 *max_num_fields* 參數。

在 3.10 版的變更: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.`**`urlunparse`**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urlsplit`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item *named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

屬性	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

警告

urlsplit() does not perform validation. See *URL parsing security* for details.

在 3.6 版的變更: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

在 3.8 版的變更: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

在 3.10 版的變更: ASCII newline and tab characters are stripped from the URL.

在 3.12 版的變更: Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)

Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

備🄵

If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*'s hostname and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with `urlsplit()` and `urlunsplit()`, removing possible *scheme* and *netloc* parts.

警告

Because an absolute URL may be passed as the `url` parameter, it is generally **not secure** to use `urljoin` with an attacker-controlled `url`. For example in, `urljoin("https://website.com/users/", username)`, if `username` can contain an absolute URL, the result of `urljoin` will be the absolute URL.

在 3.5 版的變更: Behavior updated to match the semantics defined in **RFC 3986**.

`urllib.parse.`**`urldefrag`**(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

屬性	Index	Value	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section *Structured Parse Results* for more information on the result object.

在 3.2 版的變更: Result is a structured object rather than a simple 2-tuple.

`urllib.parse.`**`unwrap`**(*url*)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

22.6.2 URL parsing security

The `urlsplit()` and `urlparse()` APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec describes what user facing web clients such as a web browser require. While **RFC 3986** is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

22.6.3 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

在 3.2 版的變更: URL parsing functions now accept ASCII encoded byte sequences

22.6.4 Structured Parse Results

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

`urllib.parse.SplitResult.`**`geturl`**`()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

`class` `urllib.parse.`**`DefragResult`**(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

在 3.2 版被加入.

`class` `urllib.parse.`**`ParseResult`**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for *urlparse()* results containing *str* data. The `encode()` method returns a *ParseResultBytes* instance.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for *urlsplit()* results containing *str* data. The `encode()` method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *bytes* data. The `decode()` method returns a *DefragResult* instance.

在 3.2 版被加入.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for *urlparse()* results containing *bytes* data. The `decode()` method returns a *ParseResult* instance.

在 3.2 版被加入.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for *urlsplit()* results containing *bytes* data. The `decode()` method returns a *SplitResult* instance.

在 3.2 版被加入.

22.6.5 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of a URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted --- its default value is `'/'`.

string may be either a *str* or a *bytes* object.

在 3.7 版的變更: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)

Like *quote()*, but also replace spaces with plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)

Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)

Replace `%xx` escapes with their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

string may be either a `str` or a `bytes` object.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

在 3.9 版的變更: *string* parameter supports bytes and str objects (previously only str).

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)

Like `unquote()`, but also replace plus signs with spaces, as required for unquoting HTML form values.

string must be a `str`.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

urllib.parse.**unquote_to_bytes**(*string*)

Replace `%xx` escapes with their single-octet equivalent, and return a `bytes` object.

string may be either a `str` or a `bytes` object.

If it is a `str`, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

Convert a mapping object or a sequence of two-element tuples, which may contain `str` or `bytes` objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the `urlopen()` function, then it should be encoded to bytes, otherwise it would result in a `TypeError`.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, `quote_plus()` is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is `quote()`, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a `str`).

To reverse this encoding process, `parse_qs()` and `parse_qsl()` are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how the `urllib.parse.urlencode()` method can be used for generating the query string of a URL or data for a POST request.

在 3.2 版的變更: *query* supports bytes and string objects.

在 3.5 版的變更: 新增 *quote_via* 參數。

在 3.14 版之後被🄵用: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

也參考

WHATWG - URL Living standard
Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers
This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's.
This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme.
Parsing requirements for mailto URL schemes.

RFC 1808 - 相對的統一資源定位器 (Relative Uniform Resource Locators)
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - 統一資源定位器 (URL, Uniform Resource Locators)
This specifies the formal syntax and semantics of absolute URLs.

22.7 `urllib.error` --- urllib.request 引發的例外類🄵

原始碼： Lib/urllib/error.py

---

urllib.error module（模組）🄵*urllib.request* 所引發的例外定義了例外 (exception) 類🄵。基礎例外類🄵是*URLError*。

下列例外會被*urllib.error* 適時引發：

exception urllib.error.**URLError**
處理程式 (handler) 在遇到問題時會引發此例外（或其衍生例外）。它是*OSError* 的一個子類🄵。

reason
此錯誤的原因。它可以是一個訊息字串或另一個例外實例。

在 3.3 版的變更: *URLError* 過去是*OSError* 的子類🄵，但現在🄵*OSError* 的🄵名。

exception urllib.error.**HTTPError**(*url*, *code*, *msg*, *hdrs*, *fp*)
雖然是一個例外（*URLError* 的一個子類🄵），*HTTPError* 也可以作🄵一個非例外的類檔案回傳值（與*urlopen()* 所回傳的物件相同）。這適用於處理特殊 HTTP 錯誤，例如🄵請求認證。

url
包含請求 URL。*filename* 屬性的🄵名。

code
一個 HTTP 狀態碼，具體定義見 **RFC 2616**。這個數值會對應到存放在*http.server.BaseHTTPRequestHandler.responses* 程式碼 dictionary 中的某個值。

reason
這通常是一個解釋本次錯誤原因的字串。🄵 *msg* 屬性的🄵名。

headers
導致*HTTPError* 的特定 HTTP 請求的 HTTP 回應 header。🄵 *hdrs* 屬性的🄵名。

在 3.4 版被加入.

fp

一個類檔案物件，可以從中讀取 HTTP 錯誤主體 (body)。

exception urllib.error.**ContentTooShortError**(*msg*, *content*)

此例外會在*urlretrieve()* 函式檢查到已下載的資料量小於期待的資料量（由 *Content-Length* header 給定）時被引發。

content

已下載（可能已被截斷）的資料。

22.8 urllib.robotparser --- robots.txt 的剖析器

原始碼： Lib/urllib/robotparser.py

此模組 (module) 提供了一個單獨的類🄵 (class) *RobotFileParser*，它可以知道某個特定 user agent（使用者代理）是否能在有發布 robots.txt 文件的網站 fetch（🄵取）特定 URL。有關 robots.txt 文件結構的更多細節，請參🄵 http://www.robotstxt.org/orig.html。

class urllib.robotparser.**RobotFileParser**(*url=''*)

此類🄵提供了一些方法可以讀取、剖析和回答關於 *url* 上的 robots.txt 文件的問題。

set_url(*url*)

設置指向 robots.txt 文件的 URL。

read()

讀取 robots.txt URL 🄵將其輸入到剖析器。

parse(*lines*)

剖析 lines 引數。

can_fetch(*useragent*, *url*)

根據從 robots.txt 文件中剖析出的規則，如果 *useragent* 被允許 fetch *url* 的話，則回傳 True。

mtime()

回傳最近一次 fetch robots.txt 文件的時間。這適用於需要定期檢查 robots.txt 文件更新情🄵的長時間運行網頁爬蟲。

modified()

將最近一次 fetch robots.txt 文件的時間設置🄵目前時間。

crawl_delay(*useragent*)

針對指定的 *useragent* 從 robots.txt 回傳 Crawl-delay 參數的值。如果此參數不存在、不適用於指定的 *useragent*，或是此參數在 robots.txt 中所指的條目含有無效語法，則回傳 None。

在 3.6 版被加入.

request_rate(*useragent*)

以*named tuple* RequestRate(requests, seconds) 的形式從 robots.txt 回傳 Request-rate 參數的🄵容。如果此參數不存在、不適用於指定的 *useragent*，或是此參數在 robots.txt 中所指的條目含有無效語法，則回傳 None。

在 3.6 版被加入.

site_maps()

以*list()* 的形式從 robots.txt 回傳 Sitemap 參數的🄵容。如果此參數不存在或此參數在 robots.txt 中所指的條目含有無效語法，則回傳 None。

在 3.8 版被加入.

下面的範例展示了*RobotFileParser* 類🄵的基本用法：

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.9 `http` --- HTTP 模組

原始碼： Lib/http/__init__.py

---

http 是一個收集了多個用於處理超文本傳輸協定 (HyperText Transfer Protocol) 之模組 (module) 的套件：

- *http.client* 是一個低階的 HTTP 協定用🄵端；對於高階的 URL 存取請使用*urllib.request*
- *http.server* 包含基於*socketserver* 的基本 HTTP 伺服器類🄵
- *http.cookies* 包含通過 cookies 實作狀態管理的工具程式 (utilities)
- *http.cookiejar* 提供了 cookies 的持續留存 (persistence)

The *http* module also defines the following enums that help you work with http related code:

class http.**HTTPStatus**

在 3.5 版被加入.

enum.IntEnum 的子類🄵，它定義了一組 HTTP 狀態碼、原理短語 (reason phrase) 以及英文長描述。

用法：

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

22.9.1 HTTP 狀態碼

http.HTTPStatus 當中，已支援🄵且有於 IANA 🄵🄵的狀態碼有：

狀態碼	列舉名懲	詳情
100	CONTINUE	HTTP Semantics **RFC 9110**, Section 15.2.1
101	SWITCHING_PROTOCOLS	HTTP Semantics **RFC 9110**, Section 15.2.2
102	PROCESSING	WebDAV **RFC 2518**，10.1 節
103	EARLY_HINTS	用於指定提示 (Indicating Hints) **RFC 8297** 的 HTTP 狀態碼
200	OK	HTTP Semantics **RFC 9110**, Section 15.3.1
201	CREATED	HTTP Semantics **RFC 9110**, Section 15.3.2
202	ACCEPTED	HTTP Semantics **RFC 9110**, Section 15.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP Semantics **RFC 9110**, Section 15.3.4
204	NO_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.5
205	RESET_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.6
206	PARTIAL_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.7
207	MULTI_STATUS	WebDAV **RFC 4918**，11.1 節
208	ALREADY_REPORTED	WebDAV 🄵結擴充 (Binding Extensions) **RFC 5842**，7.1 節（實驗性）
226	IM_USED	HTTP 中的差分編碼 **RFC 3229**，10.4.1 節
300	MULTIPLE_CHOICES	HTTP Semantics **RFC 9110**, Section 15.4.1
301	MOVED_PERMANENTLY	HTTP Semantics **RFC 9110**, Section 15.4.2
302	FOUND	HTTP Semantics **RFC 9110**, Section 15.4.3
303	SEE_OTHER	HTTP Semantics **RFC 9110**, Section 15.4.4
304	NOT_MODIFIED	HTTP Semantics **RFC 9110**, Section 15.4.5
305	USE_PROXY	HTTP Semantics **RFC 9110**, Section 15.4.6
307	TEMPORARY_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.8
308	PERMANENT_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.9
400	BAD_REQUEST	HTTP Semantics **RFC 9110**, Section 15.5.1
401	UNAUTHORIZED	HTTP Semantics **RFC 9110**, Section 15.5.2
402	PAYMENT_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.3
403	FORBIDDEN	HTTP Semantics **RFC 9110**, Section 15.5.4
404	NOT_FOUND	HTTP Semantics **RFC 9110**, Section 15.5.5
405	METHOD_NOT_ALLOWED	HTTP Semantics **RFC 9110**, Section 15.5.6
406	NOT_ACCEPTABLE	HTTP Semantics **RFC 9110**, Section 15.5.7
407	PROXY_AUTHENTICATION_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.8
408	REQUEST_TIMEOUT	HTTP Semantics **RFC 9110**, Section 15.5.9
409	CONFLICT	HTTP Semantics **RFC 9110**, Section 15.5.10
410	GONE	HTTP Semantics **RFC 9110**, Section 15.5.11
411	LENGTH_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.12
412	PRECONDITION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.13
413	CONTENT_TOO_LARGE	HTTP Semantics **RFC 9110**, Section 15.5.14
414	URI_TOO_LONG	HTTP Semantics **RFC 9110**, Section 15.5.15
415	UNSUPPORTED_MEDIA_TYPE	HTTP Semantics **RFC 9110**, Section 15.5.16
416	RANGE_NOT_SATISFIABLE	HTTP Semantics **RFC 9110**, Section 15.5.17
417	EXPECTATION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.18
418	IM_A_TEAPOT	HTCPCP/1.0 **RFC 2324**，Section 2.3.2
421	MISDIRECTED_REQUEST	HTTP Semantics **RFC 9110**, Section 15.5.20
422	UNPROCESSABLE_CONTENT	HTTP Semantics **RFC 9110**, Section 15.5.21
423	LOCKED	WebDAV **RFC 4918**，11.3 節
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**，11.4 節
425	TOO_EARLY	使用 HTTP 中的早期資料 **RFC 8470**
426	UPGRADE_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.22
428	PRECONDITION_REQUIRED	額外的 HTTP 狀態碼 **RFC 6585**
429	TOO_MANY_REQUESTS	額外的 HTTP 狀態碼 **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	額外的 HTTP 狀態碼 **RFC 6585**
451	UNAVAILABLE_FOR_LEGAL_REASONS	一個用來回報合法性障礙 (Legal Obstacles) 的 HTTP 狀態碼 **RFC 7725**
500	INTERNAL_SERVER_ERROR	HTTP Semantics **RFC 9110**, Section 15.6.1
501	NOT_IMPLEMENTED	HTTP Semantics **RFC 9110**, Section 15.6.2
502	BAD_GATEWAY	HTTP Semantics **RFC 9110**, Section 15.6.3
503	SERVICE_UNAVAILABLE	HTTP Semantics **RFC 9110**, Section 15.6.4

表格 1 – 繼續上一頁

狀態碼	列舉名懲	詳情
504	GATEWAY_TIMEOUT	HTTP Semantics RFC 9110, Section 15.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP Semantics RFC 9110, Section 15.6.6
506	VARIANT_ALSO_NEGOTIATES	HTTP 中的透明🄵容協商 (Transparent Content Negotiation) RFC 2295,
507	INSUFFICIENT_STORAGE	WebDAV RFC 4918，11.5 節
508	LOOP_DETECTED	WebDAV 🄵結擴充 RFC 5842，7.2 節（實驗性）
510	NOT_EXTENDED	一個 HTTP 擴充框架 RFC 2774，7 節（實驗性）
511	NETWORK_AUTHENTICATION_REQUIRED	額外的 HTTP 狀態碼 RFC 6585，6 節

🄵了向後相容性，列舉值也以常數形式出現在*http.client* 模組中。列舉名稱等於常數名稱（例如 http.HTTPStatus.OK 也可以是 http.client.OK）。

在 3.7 版的變更: 新增 421 MISDIRECTED_REQUEST 狀態碼。

在 3.8 版被加入: 新增 451 UNAVAILABLE_FOR_LEGAL_REASONS 狀態碼。

在 3.9 版被加入: 新增 103 EARLY_HINTS、418 IM_A_TEAPOT 與 425 TOO_EARLY 狀態碼。

在 3.13 版的變更: Implemented RFC9110 naming for status constants. Old constant names are preserved for backwards compatibility.

22.9.2 HTTP 狀態分類

在 3.12 版被加入.

The enum values have several properties to indicate the HTTP status category:

Property	Indicates that	詳情
is_informational	100 <= status <= 199	HTTP Semantics RFC 9110, Section 15
is_success	200 <= status <= 299	HTTP Semantics RFC 9110, Section 15
is_redirection	300 <= status <= 399	HTTP Semantics RFC 9110, Section 15
is_client_error	400 <= status <= 499	HTTP Semantics RFC 9110, Section 15
is_server_error	500 <= status <= 599	HTTP Semantics RFC 9110, Section 15

用法:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class http.**HTTPMethod**

在 3.11 版被加入.

enum.StrEnum 的子類🄵，它定義了一組 HTTP 方法以及英文描述。

用法:

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
```

(繼續下一頁)

(繼續上一頁)

```
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

22.9.3 HTTP 方法

http.HTTPStatus 當中，已支援🄵且有於 IANA 🄵🄵的狀態碼有：

方法	列舉名徵	詳情
GET	GET	HTTP Semantics **RFC 9110**, Section 9.3.1
HEAD	HEAD	HTTP Semantics **RFC 9110**, Section 9.3.2
POST	POST	HTTP Semantics **RFC 9110**, Section 9.3.3
PUT	PUT	HTTP Semantics **RFC 9110**, Section 9.3.4
DELETE	DELETE	HTTP Semantics **RFC 9110**, Section 9.3.5
CONNECT	CONNECT	HTTP Semantics **RFC 9110**, Section 9.3.6
OPTIONS	OPTIONS	HTTP Semantics **RFC 9110**, Section 9.3.7
TRACE	TRACE	HTTP Semantics **RFC 9110**, Section 9.3.8
PATCH	PATCH	HTTP/1.1 **RFC 5789**

22.10 `http.client` --- HTTP 協定用🄵端

原始碼： Lib/http/client.py

This module defines classes that implement the client side of the HTTP and HTTPS protocols. It is normally not used directly --- the module *urllib.request* uses it to handle URLs that use HTTP and HTTPS.

也參考

The Requests package is recommended for a higher-level HTTP client interface.

備🄵

HTTPS support is only available if Python was compiled with SSL support (through the *ssl* module).

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

The module provides the following classes:

class http.client.**HTTPConnection**(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

An *HTTPConnection* instance represents one transaction with an HTTP server. It should be instantiated by passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host,

port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

在 3.2 版的變更: 新增 *source_address*。

在 3.4 版的變更: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

在 3.7 版的變更: 新增 *blocksize* 參數。

class http.client.**HTTPSConnection**(*host*, *port=None*, *, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

A subclass of *HTTPConnection* that uses SSL for communication with secure servers. Default port is `443`. If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options.

Please read *Security considerations* for more information on best practices.

在 3.2 版的變更: 新增 *source_address*、*context* 與 *check_hostname*。

在 3.2 版的變更: This class now supports HTTPS virtual hosts if possible (that is, if *ssl.HAS_SNI* is true).

在 3.4 版的變更: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

在 3.4.3 版的變更: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

在 3.8 版的變更: This class now enables TLS 1.3 *ssl.SSLContext.post_handshake_auth* for the default *context* or when *cert_file* is passed with a custom *context*.

在 3.10 版的變更: This class now sends an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with *set_alpn_protocols()*.

在 3.12 版的變更: The deprecated *key_file*, *cert_file* and *check_hostname* parameters have been removed.

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

在 3.4 版的變更: The *strict* parameter was removed. HTTP 0.9 style "Simple Responses" are no longer supported.

This module provides the following function:

http.client.**parse_headers**(*fp*)

Parse the headers from a file pointer *fp* representing a HTTP request/response. The file has to be a *BufferedIOBase* reader (i.e. not text) and must provide a valid **RFC 2822** style header.

This function returns an instance of *http.client.HTTPMessage* that holds the header fields, but no payload (the same as *HTTPResponse.msg* and *http.server.BaseHTTPRequestHandler.headers*). After returning, the file pointer *fp* is ready to read the HTTP body.

> **備🄵**
>
> *parse_headers()* does not parse the start-line of a HTTP message; it only parses the `Name: value` lines. The file has to be ready to read these field lines, so the first line should already be consumed before calling the function.

The following exceptions are raised as appropriate:

exception http.client.**HTTPException**

The base class of the other exceptions in this module. It is a subclass of *Exception*.

exception http.client.**NotConnected**

A subclass of *HTTPException*.

exception http.client.**InvalidURL**

A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

exception http.client.**UnknownProtocol**

A subclass of *HTTPException*.

exception http.client.**UnknownTransferEncoding**

A subclass of *HTTPException*.

exception http.client.**UnimplementedFileMode**

A subclass of *HTTPException*.

exception http.client.**IncompleteRead**

A subclass of *HTTPException*.

exception http.client.**ImproperConnectionState**

A subclass of *HTTPException*.

exception http.client.**CannotSendRequest**

A subclass of *ImproperConnectionState*.

exception http.client.**CannotSendHeader**

A subclass of *ImproperConnectionState*.

exception http.client.**ResponseNotReady**

A subclass of *ImproperConnectionState*.

exception http.client.**BadStatusLine**

A subclass of *HTTPException*. Raised if a server responds with a HTTP status code that we don't understand.

exception http.client.**LineTooLong**

A subclass of *HTTPException*. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.**RemoteDisconnected**

A subclass of *ConnectionResetError* and *BadStatusLine*. Raised by *HTTPConnection.getresponse()* when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

在 3.5 版被加入: Previously, *BadStatusLine*`('')` was raised.

The constants defined in this module are:

http.client.**HTTP_PORT**

The default port for the HTTP protocol (always `80`).

http.client.**HTTPS_PORT**

The default port for the HTTPS protocol (always `443`).

http.client.**responses**

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `http.client.responses[http.client.NOT_FOUND]` is `'Not Found'`.

See *HTTP 狀態碼* for a list of HTTP status codes that are available in this module as constants.

22.10.1 HTTPConnection 物件

HTTPConnection instances have the following methods:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, *, *encode_chunked=False*)

This will send a request to the server using the HTTP request method *method* and the request URI *url*. The provided *url* must be an absolute path to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the OPTIONS or CONNECT methods).

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent; this file object should support at least the read() method. If the file object is an instance of *io.TextIOBase*, the data returned by the read() method will be encoded as ISO-8859-1, otherwise the data returned by read() is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request. A **Host header** must be provided to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the OPTIONS or CONNECT methods).

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is None, the Content-Length header is set to 0 for methods that expect a body (PUT, POST, and PATCH). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is False, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is True, the body will be chunk-encoded.

For example, to perform a GET request to https://docs.python.org/3/:

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

> **備🄵**
>
> Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

在 3.2 版的變更: *body* can now be an iterable.

在 3.6 版的變更: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

HTTPConnection.**getresponse**()

Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

> **備🄵**
>
> Note that you must have read the whole response before you can send a new request to the server.

在 3.5 版的變更: If a *ConnectionError* or subclass is raised, the *HTTPConnection* object will be ready to reconnect when a new request is sent.

HTTPConnection.**set_debuglevel**(*level*)

Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

在 3.1 版被加入.

HTTPConnection.**set_tunnel**(*host*, *port=None*, *headers=None*)

Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The *host* and *port* arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The *headers* argument should be a mapping of extra HTTP headers to send with the CONNECT request.

As HTTP/1.1 is used for HTTP CONNECT tunnelling request, as per the RFC, a HTTP `Host:` header must be provided, matching the authority-form of the request target provided as the destination for the CONNECT request. If a HTTP `Host:` header is not provided via the headers argument, one is generated and transmitted automatically.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *HTTPSConnection* constructor, and the address of the host that we eventually want to reach to the *set_tunnel()* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

在 3.2 版被加入.

在 3.12 版的變更: HTTP CONNECT tunnelling requests use protocol HTTP/1.1, upgraded from protocol HTTP/1.0. `Host:` HTTP headers are mandatory for HTTP/1.1, so one will be automatically generated and transmitted if not provided in the headers argument.

HTTPConnection.**get_proxy_response_headers**()

Returns a dictionary with the headers of the response received from the proxy server to the CONNECT request.

If the CONNECT request was not sent, the method returns `None`.

在 3.12 版被加入.

HTTPConnection.**connect**()

Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

引發一個附帶引數 `self`、`host`、`port` 的稽核事件 `http.client.connect`。

HTTPConnection.**close**()

Close the connection to the server.

HTTPConnection.**blocksize**

Buffer size in bytes for sending a file-like message body.

在 3.7 版被加入.

As an alternative to using the *request()* method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

HTTPConnection.**putheader**(*header*, *argument*[, ...])

Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*, *, *encode_chunked=False*)

Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

> **備🄵**
>
> Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

在 3.6 版的變更: Added chunked encoding support and the *encode_chunked* parameter.

HTTPConnection.**send**(*data*)

Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

引發一個附帶引數 `self`、`data` 的稽核事件 `http.client.send`。

22.10.2 HTTPResponse 物件

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

在 3.5 版的變更: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])

Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**readinto**(*b*)

Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

在 3.3 版被加入.

HTTPResponse.**getheader**(*name*, *default=None*)

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If *default* is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()

Return a list of (header, value) tuples.

HTTPResponse.**fileno**()

Return the `fileno` of the underlying socket.

HTTPResponse.**msg**

A *http.client.HTTPMessage* instance containing the response headers. *http.client.HTTPMessage* is a subclass of *email.message.Message*.

HTTPResponse.**version**

HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**url**

URL of the resource retrieved, commonly used to determine if a redirect was followed.

HTTPResponse.**headers**

Headers of the response in the form of an *email.message.EmailMessage* instance.

HTTPResponse.**status**

Status code returned by server.

HTTPResponse.**reason**

Reason phrase returned by server.

HTTPResponse.**debuglevel**

A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**

Is `True` if the stream is closed.

HTTPResponse.**geturl**()

在 3.9 版之後被🄵用: Deprecated in favor of *url*.

HTTPResponse.**info**()

在 3.9 版之後被🄵用: Deprecated in favor of *headers*.

HTTPResponse.**getcode**()

在 3.9 版之後被🄵用: Deprecated in favor of *status*.

22.10.3 范例

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
```

(繼續下一頁)

(繼續上一頁)

```
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that uses the `POST` method:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="https://bugs.python.org/issue12524">https://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side HTTP `PUT` requests are very similar to `POST` requests. The difference lies only on the server side where HTTP servers will allow resources to be created via `PUT` requests. It should be noted that custom HTTP methods are also handled in *`urllib.request.Request`* by setting the appropriate method attribute. Here is an example session that uses the `PUT` method:

```
>>> # This creates an HTTP request
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.10.4 HTTPMessage 物件

class http.client.**HTTPMessage**(*email.message.Message*)

An *`http.client.HTTPMessage`* instance holds the headers from an HTTP response. It is implemented using the *`email.message.Message`* class.

22.11 `ftplib` --- FTP 協定用🄵端

原始碼： Lib/ftplib.py

這個模組定義了*`FTP`* 類🄴和一些相關的項目。*`FTP`* 類🄴實作了 FTP 協定的用🄴端。你可以使用它來編寫能🄴執行各種 FTP 自動作業的 Python 程式，例如鏡像 (mirror) 其他 FTP 伺服器。*`urllib.request`* 模組也使用它來處理使用 FTP 的 URL。有關 FTP（檔案傳輸協定）的更多資訊，請參🄴 **RFC 959**。

預設編碼是 UTF-8，遵循 **RFC 2640**。

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄴*WebAssembly 平台*。

這是一個使用*`ftplib`* 模組的會話範例：

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')   # connect to host, default port
>>> ftp.login()                      # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')                # change into "debian" directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')            # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

22.11.1 參考

FTP 物件

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, ***, *encoding='utf-8'*)

回傳一個新的*`FTP`* 類🄴實例。

參數

- **host** (`str`) -- 要連接的主機名稱。如果有給定，`connect(host)` 會由建構函式來隱式地呼叫。
- **user** (`str`) -- The username to log in with (default: `'anonymous'`). 如果有給定，`login(host, passwd, acct)` 會由建構函式來隱式地呼叫。
- **passwd** (`str`) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **timeout** (`float` | *`None`*) -- 如*`connect()`* 的阻塞操作的超時設定，以秒🄵單位（預設：使用全域預設超時設定）。
- **source_address** (`tuple` | *`None`*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (`str`) -- The encoding for directories and filenames (default: `'utf-8'`).

FTP 類🄵支援 `with` 陳述式，例如：

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

在 3.2 版的變更: 新增了對 `with` 陳述式的支援。

在 3.3 版的變更: 新增 *source_address* 參數。

在 3.9 版的變更: 如果 *timeout* 參數設定🄵零，它將引發 *ValueError* 以防止建立非阻塞 socket。新增了 *encoding* 參數，預設值從 Latin-1 更改🄵 UTF-8 以遵循 **RFC 2640**。

`FTP` 的多個可用方法大致有分🄵兩個方向：一種用於處理文本檔案 (text files)，另一種用於二進位檔案 (binary files)。這些以在文本檔案的 `lines` 或二進位檔案的 `binary` 前使用的命令命名。

FTP 實例具有以下方法：

set_debuglevel(*level*)

將實例的偵錯級🄵設定🄵一個 *int*，這控制印出的偵錯訊息輸出量。

- `0`（預設值）：不🄵生偵錯輸出。
- `1`：會🄵生適量的偵錯輸出，通常是每個請求輸出一行。
- `2` 或更高的值：會🄵生最大量的偵錯輸出，以日🄵紀🄵下控制連🄵發送和接收的每個步驟。

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

連🄵到給定的主機 (host) 和連接埠 (port)。每個實例只應呼叫此函式一次；如果在建立 *FTP* 實例時有給定 *host* 引數，則不應呼叫它。所有其他的 `FTP` 方法只能在成功建立連🄵後使用。

參數

- **host** (`str`) -- 要連接的主機。
- **port** (`int`) -- 要連接的 TCP 連接埠（預設值：`21`，由 FTP 協定規範指定）。很少需要指定不同的連接埠號碼。
- **timeout** (`float` / *None*) -- 連🄵嘗試的超時設定，以秒🄵單位（預設：全域預設超時設定）。
- **source_address** (`tuple` / *None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

引發一個附帶引數 `self`、`host`、`port` 的稽核事件 `ftplib.connect`。

在 3.3 版的變更: 新增 *source_address* 參數。

getwelcome()

回傳伺服器🄵回應初始連🄵而發送的歡迎訊息。(此訊息有時會包含與使用者相關的免責聲明或幫助資訊。)

login(*user='anonymous'*, *passwd=''*, *acct=''*)

在以連🄵的伺服器上登入。在建立連🄵後，每個實例只應呼叫此函式一次；如果在建立 *FTP* 實例時有給定 *host* 和 *user* 引數，則不應呼叫它。大多數 FTP 命令僅在用🄵端登🄵後才允許使用

參數

- **user** (`str`) -- The username to log in with (default: `'anonymous'`).

- **passwd** (str) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (str) -- Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 for more details.

abort()

中止正在進行的檔案傳輸。使用它🄵不是都會成功，但值得一試。

sendcmd(*cmd*)

向伺服器發送一個簡單的命令字串🄵回傳回應字串。

引發一個附帶引數 `self`、`cmd` 的稽核事件 `ftplib.sendcmd`。

voidcmd(*cmd*)

向伺服器發送一個簡單的命令字串🄵處理回應。如果收到代表成功的回應狀態碼（範圍🄵 200--299 的狀態碼），則回傳回應字串，否則引發*error_reply*。

引發一個附帶引數 `self`、`cmd` 的稽核事件 `ftplib.sendcmd`。

retrbinary(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

以二進位傳輸模式 (binary transfer mode) 取得檔案。

參數

- **cmd** (str) -- 一個正確的 RETR 指令：`"RETR filename"`。
- **callback** (*callable*) -- 🄵接收到的每個資料區塊呼叫的單一參數可呼叫物件，其單一引數是以*bytes* 🄵形式的資料。
- **blocksize** (int) -- 在執行實際傳輸時所建立的低階*socket* 物件上讀取的最大分塊 (chunk) 大小。這也對應於將傳遞給 *callback* 的最大資料大小。預設🄵 8192。
- **rest** (int) -- 一個要發送到伺服器的 REST 命令。參見*transfercmd()* 方法之 *rest* 參數的文件。

retrlines(*cmd*, *callback=None*)

在初始化時以 *encoding* 參數指定的編碼來取得檔案或目🄵列表。*cmd* 要是一個正確的 RETR 命令（見*retrbinary()*）或者一個像 LIST 或 NLST 的命令（通常只是字串 `'LIST'`）。LIST 會取得檔案列表和這些檔案的相關資訊。NLST 取得檔案名稱列表。會🄵每一行以一個字串引數呼叫 *callback* 函式，其引數包含已經🄵除尾隨 CRLF 的一行。預設的 *callback* 會將各行印出到*sys.stdout*。

set_pasv(*val*)

如果 *val* 🄵真，則🄵用「被動」模式，否則禁用被動模式。被動模式預設開🄵。

storbinary(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

以二進位傳輸模式儲存檔案。

參數

- **cmd** (str) -- 一個正確的 STOR 指令：`"STOR filename"`。
- **fp** (*file object*) -- 一個檔案物件（以二進位模式開🄵），在大小🄵 *blocksize* 的區塊中使用其*read()* 方法讀取直到 EOF 來提供要儲存的資料。
- **blocksize** (int) -- 讀取區塊大小。預設🄵 8192。
- **callback** (*callable*) -- 🄵發送的每個資料區塊來呼叫的單一參數可呼叫物件，其單一引數是以*bytes* 🄵形式的資料。
- **rest** (int) -- 一個要發送到伺服器的 REST 命令。參見*transfercmd()* 方法之 *rest* 參數的文件。

在 3.2 版的變更: 新增 *rest* 參數。

storlines(*cmd*, *fp*, *callback=None*)

以行模式 (line mode) 儲存檔案。*cmd* 應是一個正確的 STOR 命令（參見*storbinary()*）。使用其*readline()* 方法從*檔案物件* *fp* （以二進位模式打開）讀取各行、直到 EOF，以提供要儲存的資料。*callback* 是可選的單參數可呼叫物件，於發送後以各行進行呼叫。

transfercmd(*cmd*, *rest=None*)

通過資料連🄵🄵動傳輸。如果傳輸🄵主動 (active) 模式，則發送 EPRT 或 PORT 命令和 *cmd* 指定的傳輸命令，🄵接受連🄵。如果伺服器是被動 (passive) 模式，則發送 EPSV 或 PASV 命令、連🄵、🄵🄵動傳輸命令。無論哪種方式，都是回傳連🄵的 socket。

如果有給定可選的 *rest*，一個 REST 命令會被發送到伺服器，🄵以 *rest* 作🄵引數。*rest* 通常是請求檔案的一個位元組偏移量 (byte offset)，告訴伺服器以請求的偏移量重新開始發送檔案的位元組，🄵跳過初始位元組。但是請注意，*transfercmd()* 方法將 *rest* 轉🄵🄵帶有初始化時指定的 *encoding* 參數的字串，但不會對字串的🄵容執行檢查。如果伺服器無法識🄵 REST 命令，則會引發*error_reply* 例外。如果發生這種情🄵，只需在🄵有 *rest* 引數的情🄵下呼叫*transfercmd()*。

ntransfercmd(*cmd*, *rest=None*)

類似於*transfercmd()*，但回傳一個帶有資料連🄵和資料預期大小的元組。如果無法計算預期大小，則回傳 None。*cmd* 和 *rest* 與*transfercmd()* 中的含義相同。

mlsd(*path=''*, *facts=[]*)

使用 MLSD 命令 (**RFC 3659**) 列出標準格式的目🄵。如果省略 *path* 則假定🄵作用於目前目🄵。*facts* 是表示所需資訊類型的字串列表（例如 `["type", "size", "perm"]`）。會回傳一個🄵生器物件，🄵每個在路徑中找到的檔案生成一個包含兩個元素的元組，第一個元素是檔案名稱，第二個元素是包含有關檔案名稱 facts 的字典。該字典的🄵容可能受 *facts* 引數限制，但不保證伺服器會回傳所有請求的 facts。

在 3.3 版被加入.

nlst(*argument*[, ...])

回傳由 NLST 命令回傳的檔案名稱列表。可選的 *argument* 是要列出的目🄵（預設🄵目前伺服器目🄵）。多個引數可用於將非標准選項傳遞給 NLST 命令。

> **備🄵**
>
> 如果你的伺服器支援該命令，*mlsd()* 會提供更好的 API。

dir(*argument*[, ...])

生成由 LIST 命令回傳的目🄵列表，🄵將其印出到標準輸出 (standard output)。可選的 *argument* 是要列出的目🄵（預設🄵目前伺服器目🄵）。有多個引數可用於將非標准選項傳遞給 LIST 命令。如果最後一個引數是一個函式，它被用作*retrlines()* 的 *callback* 函式；預設印出到*sys.stdout*。此方法回傳 None。

> **備🄵**
>
> 如果你的伺服器支援該命令，*mlsd()* 會提供更好的 API。

rename(*fromname*, *toname*)

將伺服器上的檔案 *fromname* 重新命名🄵 *toname*。

delete(*filename*)

從伺服器中🄵除名🄵 *filename* 的檔案。如果成功，回傳回應的文字，否則引發*error_perm* 權限錯誤或在其他錯誤發生時引發*error_reply*。

cwd(*pathname*)

設定伺服器上的目前目🄵。

mkd(*pathname*)

在伺服器上建立一個新目🄵。

pwd()

回傳伺服器上目前目🄵的路徑名。

rmd(*dirname*)

🄵除伺服器上名🄵 *dirname* 的目🄵。

size(*filename*)

請求伺服器上名🄵 *filename* 的檔案的大小。成功時，檔案的大小作🄵整數回傳，否則回傳 `None`。請注意，`SIZE` 命令不是標准化的，但被許多常見的伺服器實作支援。

quit()

向伺服器發送 `QUIT` 命令🄵關閉連🄵。這是關閉連🄵的「禮貌」方式，但如果伺服器對 `QUIT` 命令作出錯誤回應，它可能會引發例外。這意味著呼叫*`close()`* 方法使*`FTP`* 實例無法用於後續呼叫（見下文）。

close()

單方面關閉連🄵。這不應該使用於已經關閉的連🄵，例如在成功呼叫*`quit()`* 之後。呼叫後就不應該再次使用*`FTP`* 實例（在呼叫*`close()`* 或*`quit()`* 後，你不能通過發出另一個*`login()`* 方法重新打開連🄵）。

FTP_TLS 物件

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, ***, *context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

An *`FTP`* subclass which adds TLS support to FTP as described in **RFC 4217**. Connect to port 21 implicitly securing the FTP control connection before authenticating.

備🄵

The user must explicitly secure the data connection by calling the *`prot_p()`* method.

參數

- **host** (`str`) -- 要連接的主機名稱。如果有給定，`connect(host)` 會由建構函式來隱式地呼叫。
- **user** (`str`) -- The username to log in with (default: `'anonymous'`). 如果有給定，`login(host, passwd, acct)` 會由建構函式來隱式地呼叫。
- **passwd** (`str`) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **context** (*`ssl.SSLContext`*) -- An SSL context object which allows bundling SSL configuration options, certificates and private keys into a single, potentially long-lived, structure. Please read *Security considerations* for best practices.
- **timeout** (`float` | *`None`*) -- 如*`connect()`* 的阻塞操作的超時設定，以秒🄵單位（預設：使用全域預設超時設定）。
- **source_address** (`tuple` | *`None`*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (`str`) -- The encoding for directories and filenames (default: `'utf-8'`).

在 3.2 版被加入.

在 3.3 版的變更: 新增 *source_address* 參數。

在 3.4 版的變更: 該類🄵現在支援使用*ssl.SSLContext.check_hostname* 和 *Server Name Indication* 進行主機名 (hostname) 檢查（參見*ssl.HAS_SNI*）。

在 3.9 版的變更: 如果 *timeout* 參數設定🄵零，它將引發*ValueError* 以防止建立非阻塞 socket。新增了 *encoding* 參數，預設值從 Latin-1 更改🄵 UTF-8 以遵循 **RFC 2640**。

在 3.12 版的變更: 已🄵用的 *keyfile* 和 *certfile* 參數已被移除。

這是一個使用*FTP_TLS* 類🄵的範例會話：

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi',
↪'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore', 'libpuzzle
↪', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables', 'php-jenkins-
↪hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench', 'pincaster', 'ping',
↪'posto', 'pub', 'public', 'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance',
↪'skycache', 'sound', 'tmp', 'ucarp']
```

FTP_TLS 類🄵繼承自*FTP*，🄵另外定義了這些的方法與屬性：

ssl_version

要使用的 SSL 版本（預設🄵*ssl.PROTOCOL_SSLv23*）。

auth()

根據*ssl_version* 屬性中指定的🄵容，使用 TLS 或 SSL 設定安全控制連🄵。

在 3.4 版的變更: 該方法現在支援使用*ssl.SSLContext.check_hostname* 和 *Server Name Indication* 進行主機名檢查（參見*ssl.HAS_SNI*）。

ccc()

將控制通道恢復🄵純文本。這對於利用知道如何在不打開固定連接埠的情🄵下使用非安全 (non-secure) FTP 以處理 NAT 的防火🄵很有用。

在 3.3 版被加入.

prot_p()

設定安全資料連🄵。

prot_c()

設定明文資料 (clear text data) 連🄵。

模組變數

exception ftplib.**error_reply**

伺服器收到意外回覆時所引發的例外。

exception ftplib.**error_temp**

當收到表示暫時錯誤的錯誤碼（400--499 範圍🄵的回應狀態碼）時引發的例外。

exception ftplib.**error_perm**

當收到表示永久錯誤的錯誤碼（500--599 範圍🄵的回應狀態碼）時引發的例外。

exception ftplib.**error_proto**

當從伺服器收到不符合檔案傳輸協定回應規範的回覆時引發例外，即 1--5 範圍🄵的數字開頭。

ftplib.**all_errors**

FTP 實例方法由於 FTP 連🄵問題（相對於呼叫者的程式錯誤）而可能引發的所有例外集合（元組形式）。該集合包括上面列出的四個例外以及*OSError* 和*EOFError*。

也參考

netrc 模組
.netrc 檔案格式的剖析器。.netrc 檔案通常被 FTP 用🄵端用來在提示使用者之前載入使用者身份驗證資訊。

22.12 poplib --- POP3 協定用🄵端

原始碼： Lib/poplib.py

This module defines a class, *POP3*, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939**. The *POP3* class supports both the minimal and optional command sets from **RFC 1939**. The *POP3* class also supports the STLS command introduced in **RFC 2595** to enable encrypted communication on an already established connection.

Additionally, this module provides a class *POP3_SSL*, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the *imaplib.IMAP4* class, as IMAP servers tend to be better implemented.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

The *poplib* module provides two classes:

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

引發一個附帶引數 self、host、port 的稽核事件 poplib.connect。

引發一個附帶引數 self、line 的稽核事件 poplib.putline。其中 line 🄵即將傳送給遠端的位元組。

在 3.9 版的變更: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket.

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *, *timeout=None*, *context=None*)

This is a subclass of *POP3* that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the *POP3* constructor. *context* is an optional *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

引發一個附帶引數 self、host、port 的稽核事件 poplib.connect。

引發一個附帶引數 self、line 的稽核事件 poplib.putline。其中 line 🄵即將傳送給遠端的位元組。

在 3.2 版的變更: *context* parameter added.

在 3.4 版的變更: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

在 3.9 版的變更: If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket.

在 3.12 版的變更: The deprecated *keyfile* and *certfile* parameters have been removed.

One exception is defined as an attribute of the *poplib* module:

exception poplib.**error_proto**

Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

也參考

***imaplib* 模組**
The standard Python IMAP module.

關於 Fetchmail 的常見問題
The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

22.12.1 POP3 物件

All POP3 commands are represented by methods of the same name, in lowercase; most return the response text sent by the server.

POP3 實例擁有以下方法：

POP3.**set_debuglevel**(*level*)

Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.**getwelcome**()

Returns the greeting string sent by the POP3 server.

POP3.**capa**()

Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form `{'name': ['param'...]}`.

在 3.4 版被加入.

POP3.**user**(*username*)

Send user command, response should indicate that a password is required.

POP3.**pass_**(*password*)

Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until *quit()* is called.

POP3.**apop**(*user*, *secret*)

Use the more secure APOP authentication to log into the POP3 server.

POP3.**rpop**(*user*)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()

Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.**list**([*which*])

Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.**retr**(*which*)

Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.**dele**(*which*)

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()

Remove any deletion marks for the mailbox.

POP3.**noop**()

Do nothing. Might be used as a keep-alive.

POP3.**quit**()

Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(*which*, *howmuch*)

Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.**uidl**(*which=None*)

Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

POP3.**utf8**()

Try to switch to UTF-8 mode. Returns the server response if successful, raises *error_proto* if not. Specified in **RFC 6856**.

在 3.5 版被加入.

POP3.**stls**(*context=None*)

Start a TLS session on the active connection as specified in **RFC 2595**. This is only allowed before user authentication

context parameter is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

This method supports hostname checking via *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

在 3.4 版被加入.

Instances of *POP3_SSL* have no additional methods. The interface of this subclass is identical to its parent.

22.12.2 POP3 范例

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

22.13 `imaplib` --- IMAP4 協定用🄵端

原始碼： Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

Three classes are provided by the *imaplib* module, *IMAP4* is the base class:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

在 3.5 版的變更: Support for the `with` statement was added.

在 3.9 版的變更: 新增 *timeout* 選用參數。

Three exceptions are defined as attributes of the *IMAP4* class:

exception IMAP4.**error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, ***, *ssl_context=None*, *timeout=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

在 3.3 版的變更: 新增 *ssl_context* 參數。

在 3.4 版的變更: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

在 3.9 版的變更: 新增 *timeout* 選用參數。

在 3.12 版的變更: The deprecated *keyfile* and *certfile* parameters have been removed.

The second subclass allows for connections created by a child process:

class imaplib.**IMAP4_stream**(*command*)

This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

imaplib.**Internaldate2tuple**(*datestr*)

Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* tuple or `None` if the string has wrong format.

imaplib.**Int2AP**(*num*)

Converts an integer into a bytes representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)

Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

也參考

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

22.13.1 IMAP4 物件

All IMAP4rev1 commands are represented by methods of the same name, either uppercase or lowercase.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Most commands return a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

IMAP4 實例擁有以下方法:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)

Append *message* to named mailbox.

IMAP4.**authenticate**(*mechanism*, *authobject*)

Authenticate command --- requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

在 3.5 版的變更: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

IMAP4.**check**()

Checkpoint mailbox on server.

IMAP4.**close**()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

IMAP4.**copy**(*message_set*, *new_mailbox*)

Copy *message_set* messages onto end of *new_mailbox*.

IMAP4.**create**(*mailbox*)

Create new mailbox named *mailbox*.

IMAP4.**delete**(*mailbox*)

Delete old mailbox named *mailbox*.

IMAP4.**deleteacl**(*mailbox*, *who*)

Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.**enable**(*capability*)

Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see **RFC 6855**).

在 3.5 版被加入: The *`enable()`* method itself, and **RFC 6855** support.

IMAP4.**expunge**()

Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)

Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)

Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)

Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**getquota**(*root*)

Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot**(*mailbox*)

Get the list of `quota roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**idle**(*duration=None*)

Return an `Idler`: an iterable context manager implementing the IMAP4 `IDLE` command as defined in **RFC 2177**.

The returned object sends the `IDLE` command when activated by the `with` statement, produces IMAP untagged responses via the *iterator* protocol, and sends `DONE` upon context exit.

All untagged responses that arrive after sending the `IDLE` command (including any that arrive before the server acknowledges the command) will be available via iteration. Any leftover responses (those not iterated in the `with` context) can be retrieved in the usual way after `IDLE` ends, using *IMAP4.response()*.

Responses are represented as `(type, [data, ...])` tuples, as described in *IMAP4 Objects*.

The *duration* argument sets a maximum duration (in seconds) to keep idling, after which any ongoing iteration will stop. It can be an *int* or *float*, or `None` for no time limit. Callers wishing to avoid inactivity timeouts on servers that impose them should keep this at most 29 minutes (1740 seconds). Requires a socket connection; *duration* must be `None` on *IMAP4_stream* connections.

```
>>> with M.idle(duration=29 * 60) as idler:
...     for typ, data in idler:
...         print(typ, data)
...
EXISTS [b'1']
RECENT [b'1']
```

Idler.**burst**(*interval=0.1*)

Yield a burst of responses no more than *interval* seconds apart (expressed as an *int* or *float*).

This *generator* is an alternative to iterating one response at a time, intended to aid in efficient batch processing. It retrieves the next response along with any immediately available subsequent responses. (For example, a rapid series of `EXPUNGE` responses after a bulk delete.)

Requires a socket connection; does not work on *IMAP4_stream* connections.

```
>>> with M.idle() as idler:
...     # get a response and any others following by < 0.1 seconds
...     batch = list(idler.burst())
...     print(f'processing {len(batch)} responses...')
...     print(batch)
...
processing 3 responses...
[('EXPUNGE', [b'2']), ('EXPUNGE', [b'1']), ('RECENT', [b'0'])]
```

小訣竅

The `IDLE` context's maximum duration, as passed to *IMAP4.idle()*, is respected when waiting for the first response in a burst. Therefore, an expired `Idler` will cause this generator to return immediately without producing anything. Callers should consider this if using it in a loop.

備🄵

The iterator returned by *IMAP4.idle()* is usable only within a `with` statement. Before or after that context, unsolicited responses are collected internally whenever a command finishes, and can be retrieved with *IMAP4.response()*.

備🄵

The `Idler` class name and structure are internal interfaces, subject to change. Calling code can rely on its context management, iteration, and public method to remain stable, but should not subclass, instantiate, compare, or otherwise directly reference the class.

在 3.14 版被加入.

`IMAP4.`**`list`**`(`[*directory*[, *pattern*]]`)`

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

`IMAP4.`**`login`**`(`*user*, *password*`)`

Identify the client using a plaintext password. The *password* will be quoted.

`IMAP4.`**`login_cram_md5`**`(`*user*, *password*`)`

Force use of `CRAM-MD5` authentication when identifying the client to protect the password. Will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

在 3.14 版的變更: An *`IMAP4.error`* is raised if MD5 support is not available.

`IMAP4.`**`logout`**`()`

Shutdown connection to server. Returns server `BYE` response.

在 3.8 版的變更: The method no longer ignores silently arbitrary exceptions.

`IMAP4.`**`lsub`**`(`*directory*='""', *pattern*='*'`)`

List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

`IMAP4.`**`myrights`**`(`*mailbox*`)`

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

`IMAP4.`**`namespace`**`()`

Returns IMAP namespaces as defined in **RFC 2342**.

`IMAP4.`**`noop`**`()`

Send `NOOP` to server.

`IMAP4.`**`open`**`(`*host*, *port*, *timeout=None*`)`

Opens socket to *port* at *host*. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used. Also note that if the *timeout* parameter is set to be zero, it will raise a *`ValueError`* to reject creating a non-blocking socket. This method is implicitly called by the *`IMAP4`* constructor. The connection objects established by this method will be used in the *`IMAP4.read()`*, *`IMAP4.readline()`*, *`IMAP4.send()`*, and *`IMAP4.shutdown()`* methods. You may override this method.

引發一個附帶引數 `self`、`host`、`port` 的稽核事件 `imaplib.open`。

在 3.9 版的變更: 新增 *timeout* 參數。

`IMAP4.`**`partial`**`(`*message_num*, *message_part*, *start*, *length*`)`

Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

`IMAP4.`**`proxyauth`**`(`*user*`)`

Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

`IMAP4.`**`read`**`(`*size*`)`

Reads *size* bytes from the remote server. You may override this method.

`IMAP4.`**`readline`**`()`

Reads one line from the remote server. You may override this method.

IMAP4.**recent**()

Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)

Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)

Return data for response *code* if received, or `None`. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])

Search mailbox for matching messages. *charset* may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be `None` if the `UTF8=ACCEPT` capability was enabled using the *enable()* command.

範例:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)

Select a mailbox. Returned data is the count of messages in *mailbox* (`EXISTS` response). The default *mailbox* is `'INBOX'`. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)

Sends `data` to the remote server. You may override this method.

引發一個附帶引數 `self`、`data` 的稽核事件 `imaplib.send`。

IMAP4.**setacl**(*mailbox*, *who*, *what*)

Set an `ACL` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])

Set `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**setquota**(*root*, *limits*)

Set the `quota` *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()

Close connection established in `open`. This method is implicitly called by *IMAP4.logout()*. You may override this method.

IMAP4.**socket**()

Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])

The `sort` command is a variant of `search` with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid sort` command which corresponds to `sort` the way that `uid search` corresponds to `search`. The `sort` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an `IMAP4rev1` extension command.

IMAP4.**starttls**(*ssl_context=None*)

Send a STARTTLS command. The *ssl_context* argument is optional and should be a *ssl.SSLContext* object. This will enable encryption on the IMAP connection. Please read *Security considerations* for best practices.

在 3.2 版被加入.

在 3.4 版的變更: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

IMAP4.**status**(*mailbox*, *names*)

Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

備🄵

Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib still continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)

Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])

The thread command is a variant of search with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid thread command which corresponds to thread the way that uid search corresponds to search. The thread command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an IMAP4rev1 extension command.

IMAP4.**uid**(*command*, *arg*[, ...])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.**unsubscribe**(*mailbox*)

Unsubscribe from old mailbox.

IMAP4.**unselect**()

imaplib.IMAP4.unselect() frees server's resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as *imaplib.IMAP4.close()*, except that no messages are permanently removed from the currently selected mailbox.

在 3.9 版被加入.

IMAP4.**xatom**(*name*[, ...])

Allow simple extension commands notified by server in CAPABILITY response.

The following attributes are defined on instances of *IMAP4*:

IMAP4.**PROTOCOL_VERSION**

The most recent supported protocol in the CAPABILITY response from the server.

IMAP4.**debug**

Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.

IMAP4.**utf8_enabled**

Boolean value that is normally False, but is set to True if an *enable()* command is successfully issued for the UTF8=ACCEPT capability.

在 3.5 版被加入.

22.13.2 IMAP4 范例

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

22.14 smtplib --- SMTP 協定用🄵端

原始碼： Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional *host* and *port* parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt

(if not specified, the global default timeout setting will be used). If the timeout expires, `TimeoutError` is raised. The optional *source_address* parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple `(host, port)`, for the socket to bind to as its source address before connecting. If omitted (or if *host* or *port* are `''` and/or `0` respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, `sendmail()`, and `SMTP.quit()` methods. An example is included below.

The `SMTP` class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

所有指令都會引發一個附帶引數 `self`、`data` 的稽核事件 `smtplib.SMTP.send`，其中 `data` 🄵即將傳送至遠端的位元組。

在 3.3 版的變更: Support for the `with` statement was added.

在 3.3 版的變更: 新增 source_address 引數。

在 3.5 版被加入: The SMTPUTF8 extension (**RFC 6531**) is now supported.

在 3.9 版的變更: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, ***, [*timeout*,]*context=None*, *source_address=None*)

An `SMTP_SSL` instance behaves exactly the same as instances of `SMTP`. `SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the `SMTP` class. *context*, also optional, can contain a `SSLContext` and allows configuring various aspects of the secure connection. Please read *Security considerations* for best practices.

在 3.3 版的變更: 新增 *context*。

在 3.3 版的變更: 新增 *source_address* 引數。

在 3.4 版的變更: The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

在 3.9 版的變更: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket

在 3.12 版的變更: The deprecated *keyfile* and *certfile* parameters have been removed.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments *local_hostname* and *source_address* have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

在 3.9 版的變更: 新增 *timeout* 選用參數。

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**

Subclass of *OSError* that is the base exception class for all the other exceptions provided by this module.

在 3.4 版的變更: SMTPException became subclass of *OSError*

exception smtplib.**SMTPServerDisconnected**

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the *SMTP* instance before connecting it to a server.

exception smtplib.**SMTPResponseException**

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**

Sender address refused. In addition to the attributes set by on all *SMTPResponseException* exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as *SMTP.sendmail()* returns.

exception smtplib.**SMTPDataError**

The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**

Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**

The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**

The command or option attempted is not supported by the server.

在 3.5 版被加入.

exception smtplib.**SMTPAuthenticationError**

SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

也參考

RFC 821 - Simple Mail Transfer Protocol
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

22.14.1 SMTP 物件

SMTP 實例擁有以下方法:

SMTP.**set_debuglevel**(*level*)

Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

在 3.5 版的變更: Added debuglevel 2.

SMTP.**docmd**(*cmd*, *args=''*)

Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**(*host='localhost'*, *port=0*)

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

引發一個附帶引數 `self`、`host`、`port` 的稽核事件 `smtplib.connect`。

SMTP.**helo**(*name=''*)

Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**(*name=''*)

Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to `True` or `False` depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()

This method calls *ehlo()* and/or *helo()* if there has been no previous `EHLO` or `HELO` command this session. It tries ESMTP `EHLO` first.

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTP.**has_extn**(*name*)

Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)

Check the validity of an address on this server using SMTP `VRFY`. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

> **備🄵**
>
> Many sites disable SMTP `VRFY` in order to foil spammers.

SMTP.**login**(*user*, *password*, ***, *initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous `EHLO` or `HELO` command this session, this method tries

ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError
The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError
The server didn't accept the username/password combination.

SMTPNotSupportedError
The AUTH command is not supported by the server.

SMTPException
No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

在 3.5 版的變更: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

```
data = authobject(challenge=None)
```

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns None, or if *initial_response_ok* is false, authobject() will be called to process the server's challenge response; the *challenge* argument it is passed will be a bytes. It should return ASCII str *data* that will be base64 encoded and sent to the server.

The SMTP class provides authobjects for the CRAM-MD5, PLAIN, and LOGIN mechanisms; they are named SMTP.auth_cram_md5, SMTP.auth_plain, and SMTP.auth_login respectively. They all require that the user and password properties of the SMTP instance are set to appropriate values.

User code does not normally need to call auth directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. auth is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

在 3.5 版被加入.

SMTP.**starttls**(*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

在 3.12 版的變更: The deprecated *keyfile* and *certfile* parameters have been removed.

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTPNotSupportedError
The server does not support the STARTTLS extension.

RuntimeError
SSL/TLS support is not available to your Python interpreter.

在 3.3 版的變更: 新增 *context*。

在 3.4 版的變更: The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see *HAS_SNI*).

在 3.5 版的變更: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

`SMTP.`**`sendmail`**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in `MAIL FROM` commands as *mail_options*. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

> **備🄵**
>
> The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused
All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused
The server didn't accept the *from_addr*.

SMTPDataError
The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError
SMTPUTF8 was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

在 3.2 版的變更: *msg* may be a byte string.

在 3.5 版的變更: SMTPUTF8 support added, and *SMTPNotSupportedError* may be raised if SMTPUTF8 is specified but the server does not support it.

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a Message object.

If *from_addr* is None or *to_addrs* is None, send_message fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

send_message serializes *msg* using *BytesGenerator* with \r\n as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, send_message does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise SMTPUTF8 support, an *SMTPNotSupportedError* is raised. Otherwise the Message is serialized with a clone of its *policy* with the *utf8* attribute set to True, and SMTPUTF8 and BODY=8BITMIME are added to *mail_options*.

在 3.2 版被加入.

在 3.5 版被加入: Support for internationalized addresses (SMTPUTF8).

SMTP.**quit**()

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

22.14.2 SMTP 范例

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly:

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}", ""]
while True:
    try:
        line = input()
    except EOFError:
        break
```

(繼續下一頁)

(繼續上一頁)

```
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

備🄵

In general, you will want to use the *email* package's features to construct an email message, which you can then send via *send_message()*; see *email: 範例*.

22.15 uuid --- RFC 9562 定義的 UUID 物件

原始碼： Lib/uuid.py

This module provides immutable *UUID* objects (the *UUID* class) and *functions* for generating UUIDs corresponding to a specific UUID version as specified in **RFC 9562** (which supersedes **RFC 4122**), for example, *uuid1()* for UUID version 1, *uuid3()* for UUID version 3, and so on. Note that UUID version 2 is deliberately omitted as it is outside the scope of the RFC.

如果你只需要一個唯一的 ID，你應該呼叫*uuid1()* 或*uuid4()*。需要注意的是，*uuid1()* 可能會危害隱私，因🄵它會建立一個包含了電腦網路位址的 UUID。而*uuid4()* 則會建立一個隨機的 UUID。

根據底層平台的支援情🄵，*uuid1()* 可能會也可能不會回傳一個「安全的」UUID。安全的 UUID 是使用同步方法生成的，以確保不會有兩個行程取得到相同的 UUID。所有*UUID* 的實例都有一個*is_safe* 屬性，該屬性使用下面這個列舉來傳遞有關 UUID 安全性的任何資訊：

class uuid.**SafeUUID**

在 3.7 版被加入.

safe

該 UUID 是由平台以多行程安全的 (multiprocessing-safe) 方式生成的。

unsafe

該 UUID 不是以多行程安全的方式生成的。

unknown

該平台不提供 UUID 是否🄵安全生成的資訊。

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)

從以下其中一種引數來建立 UUID：由 32 個十六進位的數字組成的字串、以大端順序 (big-endian) 排列的 16 個位元組組成的字串作🄵 *bytes* 引數、以小端順序 (little-endian) 排列的 16 個位元組組成的字串作🄵 *bytes_le* 引數、由 6 個整數（32 位元的 *time_low*、16 位元的 *time_mid*、16 位元的 *time_hi_version*、8 位元的 *clock_seq_hi_variant*、8 位元的 *clock_seq_low*、48 位元的 *node*）組成的元組 (tuple) 作🄵 *fields* 引數，或者是單一的 128 位元整數作🄵 *int* 引數。當給定由十六進位的數字組成的字串時，大括號、連字符號和 URN 前綴都是可以選用的。例如，以下這些運算式都會🄵生相同的 UUID：

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

必須正好給定其中一個引數 *hex*、*bytes*、*bytes_le*、*fields* 或 *int*。*version* 引數是選用的；如果給定了該引數，生成的 UUID 將根據 **RFC 9562** 設置其變體 (variant) 和版本號，覆蓋掉給定的 *hex*、*bytes*、*bytes_le*、*fields* 或 *int* 中的位元。

UUID 物件之間的比較是透過比較它們的*UUID.int* 屬性。與非 UUID 的物件進行比較會引發*TypeError*。

`str(uuid)` 會回傳一個像是 `12345678-1234-5678-1234-567812345678` 形式的字串，其中 32 個十六進位的數字代表 UUID。

UUID 實例有以下唯讀的屬性：

UUID.bytes

UUID 以 16 位元組的字串表示（包含 6 個整數欄位，按照大端位元組順序排列）。

UUID.bytes_le

UUID 以 16 位元組的字串表示（其中 *time_low*、*time_mid* 和 *time_hi_version* 按照小端位元組順序排列）。

UUID.fields

UUID 以 6 個整數欄位所組成的元組表示，也可以看作有 6 個個🄵屬性和 2 個衍生屬性：

欄位	意義
UUID.time_low	UUID 的前 32 位元。只與版本 1 有關。
UUID.time_mid	UUID 接下來的 16 位元。只與版本 1 有關。
UUID.time_hi_version	UUID 接下來的 16 位元。只與版本 1 有關。
UUID.clock_seq_hi_variant	UUID 接下來的 8 位元。只與版本 1 和 6 有關。
UUID.clock_seq_low	UUID 接下來的 8 位元。只與版本 1 和 6 有關。
UUID.node	UUID 最後的 48 位元。只與版本 1 有關。
UUID.time	The 60-bit timestamp as a count of 100-nanosecond intervals since Gregorian epoch (1582-10-15 00:00:00) for versions 1 and 6, or the 48-bit timestamp in milliseconds since Unix epoch (1970-01-01 00:00:00) for version 7.
UUID.clock_seq	14 位元的序列號。只與版本 1 和 6 有關。

UUID.**hex**
UUID 以 32 個小寫十六進位字元組成的字串表示。

UUID.**int**
UUID 以 128 位元的整數表示。

UUID.**urn**
UUID 以 **RFC 9562** 中指定的 URN 形式表示。

UUID.**variant**
UUID 的變體，🄵定 UUID 🄵部的🄵局 (layout)，是*RESERVED_NCS*、*RFC_4122*、*RESERVED_MICROSOFT* 或*RESERVED_FUTURE* 其中一個常數。

UUID.**version**
UUID 的版本號（1 到 8，只有當變體是*RFC_4122* 時才有意義）。

在 3.14 版的變更: 新增了 UUID 版本 6、7 和 8。

UUID.**is_safe**
SafeUUID 的列舉，表示平台是否以多行程安全的方式🄵生 UUID。

在 3.7 版被加入.

uuid 模組定義了以下函式:

uuid.**getnode**()
取得 48 位元正整數形式的硬體位址。第一次執行時，有可能會🄵動一個獨立的程式，這也許會相當耗時。如果所有取得硬體位址的嘗試都失敗，我們會根據 **RFC 4122** 中的建議，使用一個 48 位元的隨機數，其中群播位元 (multicast bit)（第一個八位元組的最低有效位）設置🄵 1。「硬體位址」指的是網路介面 (network interface) 的 MAC 位址。在具有多個網路介面的機器上，將優先選擇通用管理 (universally administered) 的 MAC 位址（即第一個八位元組的第二個最低有效位是 未設置的），而不是本地管理 (locally administered) 的 MAC 位址，除此之外不保證任何選擇的順序。

在 3.7 版的變更: 通用管理的 MAC 位址優於本地管理的 MAC 位址，因🄵前者保證是全球唯一的，而後者不是。

uuid.**uuid1**(*node=None*, *clock_seq=None*)
Generate a UUID from a host ID, sequence number, and the current time according to **RFC 9562, §5.1**.

When *node* is not specified, *getnode()* is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

uuid.**uuid3**(*namespace*, *name*)
根據 **RFC 9562, §5.3**，基於命名空間識🄵碼 (namespace identifier)（一個 UUID）和名稱（一個*bytes* 物件或使用 UTF-8 編碼的字串）的 MD5 hash 來生成 UUID。

uuid.**uuid4**()
Generate a random UUID in a cryptographically-secure method according to **RFC 9562, §5.4**.

uuid.**uuid5**(*namespace*, *name*)
根據 **RFC 9562, §5.5**，基於命名空間識🄵碼（一個 UUID）和名稱（一個*bytes* 物件或使用 UTF-8 編碼的字串）的 SHA-1 hash 來生成 UUID。

uuid.**uuid6**(*node=None*, *clock_seq=None*)
Generate a UUID from a sequence number and the current time according to **RFC 9562, §5.6**.

This is an alternative to *uuid1()* to improve database locality.

When *node* is not specified, *getnode()* is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

在 3.14 版被加入.

uuid.**uuid7**()

Generate a time-based UUID according to **RFC 9562, §5.7**.

For portability across platforms lacking sub-millisecond precision, UUIDs produced by this function embed a 48-bit timestamp and use a 42-bit counter to guarantee monotonicity within a millisecond.

在 3.14 版被加入.

uuid.**uuid8**(*a=None*, *b=None*, *c=None*)

Generate a pseudo-random UUID according to **RFC 9562, §5.8**.

When specified, the parameters *a*, *b* and *c* are expected to be positive integers of 48, 12 and 62 bits respectively. If they exceed their expected bit count, only their least significant bits are kept; non-specified arguments are substituted for a pseudo-random integer of appropriate size.

By default, *a*, *b* and *c* are not generated by a cryptographically secure pseudo-random number generator (CSPRNG). Use *uuid4()* when a UUID needs to be used in a security-sensitive context.

在 3.14 版被加入.

uuid 模組🄵*uuid3()* 或*uuid5()* 定義了以下的命名空間識🄵碼。

uuid.**NAMESPACE_DNS**

當指定這個命名空間時，*name* 字串是一個完整網域名稱 (fully qualified domain name)。

uuid.**NAMESPACE_URL**

當指定這個命名空間時，*name* 字串是一個 URL。

uuid.**NAMESPACE_OID**

當指定這個命名空間時，*name* 字串是一個 ISO OID。

uuid.**NAMESPACE_X500**

當指定這個命名空間時，*name* 字串是以 DER 或文字輸出格式表示的 X.500 DN。

uuid 模組🄵*variant* 屬性的可能值定義了以下常數：

uuid.**RESERVED_NCS**

保留供 NCS 相容性使用。

uuid.**RFC_4122**

Specifies the UUID layout given in **RFC 4122**. This constant is kept for backward compatibility even though **RFC 4122** has been superseded by **RFC 9562**.

uuid.**RESERVED_MICROSOFT**

保留供 Microsoft 相容性使用。

uuid.**RESERVED_FUTURE**

保留供未來定義使用。

uuid 模組定義了特殊的 Nil 和 Max UUID 值：

uuid.**NIL**

A special form of UUID that is specified to have all 128 bits set to zero according to **RFC 9562, §5.9**.

在 3.14 版被加入.

uuid.**MAX**

A special form of UUID that is specified to have all 128 bits set to one according to **RFC 9562, §5.10**.

在 3.14 版被加入.

也參考

RFC 9562 - 通用唯一辨識碼 (UUID, Universally Unique IDentifier) 的 URN 命名空間
這個規範定義了 UUID 的統一資源名稱 (Uniform Resource Name) 命名空間、UUID 的🄵部格式和生成 UUID 的方法。

22.15.1 命令列的用法

在 3.12 版被加入.

uuid 模組可以在命令列下作🄵🄵本來執行。

```
python -m uuid [-h] [-u {uuid1,uuid3,uuid4,uuid5,uuid6,uuid7,uuid8}] [-n NAMESPACE] [-N NAME]
```

可以接受以下選項:

-h, --help
顯示幫助訊息🄴退出。

-u <uuid>

--uuid <uuid>
指定要用來生成 UUID 的函式名稱。預設使用*uuid4()*。

在 3.14 版的變更: Allow generating UUID versions 6, 7 and 8.

-n <namespace>

--namespace <namespace>
該命名空間是一個 `UUID` 或 `@ns`, 其中 `ns` 是指知名預定義 UUID 的命名空間名稱, 例如 `@dns`、`@url`、`@oid` 和 `@x500`。只有*uuid3()* / *uuid5()* 函式會需要。

-N <name>

--name <name>
用於生成 uuid 的名稱。只有*uuid3()* / *uuid5()* 函式會需要。

-C <num>

--count <num>
生成 *num* 個新 UUID。

在 3.14 版被加入.

22.15.2 范例

以下是一些*uuid* 模組的典型使用範例:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
```

(繼續下一頁)

(繼續上一頁)

```
>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

>>> # get the Nil UUID
>>> uuid.NIL
UUID('00000000-0000-0000-0000-000000000000')

>>> # get the Max UUID
>>> uuid.MAX
UUID('ffffffff-ffff-ffff-ffff-ffffffffffff')

>>> # same as UUIDv1 but with fields reordered to improve DB locality
>>> uuid.uuid6()
UUID('1f0799c0-98b9-62db-92c6-a0d365b91053')

>>> # get UUIDv7 creation (local) time as a timestamp in milliseconds
>>> u = uuid.uuid7()
>>> u.time
1743936859822

>>> # get UUIDv7 creation (local) time as a datetime object
>>> import datetime as dt
>>> dt.datetime.fromtimestamp(u.time / 1000)
datetime.datetime(...)

>>> # make a UUID with custom blocks
>>> uuid.uuid8(0x12345678, 0x9abcdef0, 0x11223344)
UUID('00001234-5678-8ef0-8000-000011223344')
```

22.15.3 命令列的范例

以下是一些*uuid*命令列介面的典型使用範例：

```
# 生成一個隨機的 UUID - 預設使用 uuid4()
$ python -m uuid

# 使用 uuid1() 生成一組 UUID
$ python -m uuid -u uuid1

# 使用 uuid5 生成一組 UUID
$ python -m uuid -u uuid5 -n @url -N example.com

# 生成 42 個隨機 UUID
$ python -m uuid -C 42
```

22.16 `socketserver` --- 用於網路伺服器的框架

原始碼： Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

There are four basic concrete server classes:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

22.16.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
```

(繼續下一頁)

(繼續上一頁)

```
      v
+-----------+         +--------------------+
| UDPServer |-------->| UnixDatagramServer |
+-----------+         +--------------------+
```

Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* --- the only difference between an IP and a Unix server is the address family.

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

block_on_close

ForkingMixIn.server_close waits until all child processes complete, except if *block_on_close* attribute is `False`.

ThreadingMixIn.server_close waits until all non-daemon threads complete, except if *block_on_close* attribute is `False`.

max_children

Specify how many child processes will exist to handle requests at a time for *ForkingMixIn*. If the limit is reached, new requests will wait until one child process has finished.

daemon_threads

For *ThreadingMixIn* use daemonic threads by setting *ThreadingMixIn.daemon_threads* to `True` to not wait until threads complete.

在 3.7 版的變更: *ForkingMixIn.server_close* and *ThreadingMixIn.server_close* now waits until all child processes and non-daemonic threads complete. Add a new *ForkingMixIn.block_on_close* class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

class socketserver.**ForkingUnixStreamServer**

class socketserver.**ForkingUnixDatagramServer**

class socketserver.**ThreadingUnixStreamServer**

class socketserver.**ThreadingUnixDatagramServer**

These classes are pre-defined using the mix-in classes.

在 3.12 版被加入: The `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes were added.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process

would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled -- which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used).

22.16.2 Server Objects

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

在 3.3 版的變更: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

在 3.3 版被加入.

shutdown()

Tell the *serve_forever()* loop to stop and wait until it does. *shutdown()* must be called while *serve_forever()* is running in a different thread otherwise it will deadlock.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET*, *socket.AF_INET6*, and *socket.AF_UNIX*. Subclass the TCP or UDP server classes in this module with class attribute `address_family = AF_INET6` set if you want IPv6 server classes.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address

Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type

The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout

Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)

Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()

Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)

This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

在 3.6 版的變更: Now only called for exceptions derived from the *Exception* class.

handle_timeout()

This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)

Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()

Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)

Must return a Boolean value; if the value is `True`, the request will be processed, and if it's `False`, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns `True`.

在 3.6 版的變更: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `server_close()`.

22.16.3 Request Handler Objects

class socketserver.**BaseRequestHandler**

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new `handle()` method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the `handle()` method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `request`; the client address as `client_address`; and the server instance as `server`, in case it needs access to per-server information.

The type of `request` is different for datagram or stream services. For stream services, `request` is a socket object; for datagram services, `request` is a pair of string and socket.

finish()

Called after the `handle()` method to perform any clean-up actions required. The default implementation does nothing. If `setup()` raises an exception, this function will not be called.

request

The *new* `socket.socket` object to be used to communicate with the client.

client_address

Client address returned by `BaseServer.get_request()`.

server

`BaseServer` object used for handling the request.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

These `BaseRequestHandler` subclasses override the `setup()` and `finish()` methods, and provide `rfile` and `wfile` attributes.

rfile

A file object from which receives the request is read. Support the `io.BufferedIOBase` readable interface.

wfile

A file object to which the reply is written. Support the `io.BufferedIOBase` writable interface

在 3.6 版的變更: `wfile` also supports the `io.BufferedIOBase` writable interface.

22.16.4 范例

socketserver.TCPServer 范例

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        pieces = [b'']
        total = 0
        while b'\n' not in pieces[-1] and total < 10_000:
            pieces.append(self.request.recv(2000))
            total += len(pieces[-1])
        self.data = b''.join(pieces)
        print(f"Received from {self.client_address[0]}:")
        print(self.data.decode("utf-8"))
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())
        # after we return, the socket will be closed.

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler.
        # We can now use e.g. readline() instead of raw recv() calls.
        # We limit ourselves to 10000 bytes to avoid abuse by the sender.
        self.data = self.rfile.readline(10000).rstrip()
        print(f"{self.client_address[0]} wrote:")
        print(self.data.decode("utf-8"))
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the first handler had to use a `recv()` loop to accumulate data until a newline itself. If it had just used a single `recv()` without the loop it would just have returned what has been received so far from the client. TCP is stream based: data arrives in the order it was sent, but there no correlation between client `send()` or `sendall()` calls and the number of `recv()` calls on the server required to receive it.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
```

(繼續下一頁)

(繼續上一頁)

```
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data, "utf-8"))
    sock.sendall(b"\n")

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

`socketserver.UDPServer` 范例

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print(f"{self.client_address[0]} wrote:")
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)
```

(繼續下一頁)

(繼續上一頁)

```
        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

22.17 http.server —HTTP 伺服器

原始碼： Lib/http/server.py

此模組定義了用於實作 HTTP 伺服器的類🄴。

> **⚠ 警告**
>
> *http.server* is not recommended for production. It only implements *basic security checks*.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄴*WebAssembly 平台*。

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

This class builds on the *TCPServer* class by storing the server address as instance variables named `server_name` and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

在 3.7 版被加入.

class http.server.**HTTPSServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, ***, *certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

Subclass of *HTTPServer* with a wrapped socket using the *ssl* module. If the *ssl* module is not available, instantiating a `HTTPSServer` object fails with a *RuntimeError*.

The *certfile* argument is the path to the SSL certificate chain file, and the *keyfile* is the path to file containing the private key.

A *password* can be specified for files protected and wrapped with PKCS#8, but beware that this could possibly expose hardcoded passwords in clear.

> **也參考**
>
> See `ssl.SSLContext.load_cert_chain()` for additional information on the accepted values for *certfile*, *keyfile* and *password*.

When specified, the *alpn_protocols* argument must be a sequence of strings specifying the "Application-Layer Protocol Negotiation" (ALPN) protocols supported by the server. ALPN allows the server and the client to negotiate the application protocol during the TLS handshake.

By default, it is set to `["http/1.1"]`, meaning the server supports HTTP/1.1.

在 3.14 版被加入.

class http.server.**ThreadingHTTPSServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, ***, *certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

This class is identical to `HTTPSServer` but uses threads to handle requests by inheriting from `ThreadingMixIn`. This is analogous to `ThreadingHTTPServer` only using `HTTPSServer`.

在 3.14 版被加入.

The `HTTPServer`, `ThreadingHTTPServer`, `HTTPSServer` and `ThreadingHTTPSServer` must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). `BaseHTTPRequestHandler` provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

`BaseHTTPRequestHandler` 有以下實例變數:

client_address

Contains a tuple of the form `(host, port)` referring to the client's address.

server

Contains the server instance.

close_connection

Boolean that should be set before `handle_one_request()` returns, indicating if another request may be expected, or if the connection should be shut down.

requestline

Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by `handle_one_request()`. If no valid request line was processed, it should be set to the empty string.

command

Contains the command (request type). For example, `'GET'`.

path

Contains the request path. If query component of the URL is present, then `path` includes the query. Using the terminology of **RFC 3986**, `path` here includes `hier-part` and the `query`.

request_version

Contains the version string from the request. For example, `'HTTP/1.0'`.

headers

Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The *parse_headers()* function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile

An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile

Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

在 3.6 版的變更: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler 擁有以下屬性:

server_version

Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version

Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format

Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type

Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version

Specifies the HTTP version to which the server is conformant. It is sent in responses to let the client know the server's communication capabilities for future requests. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass

Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to *http.client.HTTPMessage*.

responses

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

BaseHTTPRequestHandler 實例擁有以下方法:

handle()

Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()

This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()

When an HTTP/1.1 conformant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can choose to send `417 Expectation Failed` as a response header and `return False`.

在 3.2 版被加入.

send_error(*code*, *message=None*, *explain=None*)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

在 3.4 版的變更: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

在 3.3 版的變更: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)

Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

在 3.2 版的變更: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)

Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

在 3.2 版被加入.

end_headers()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

在 3.2 版的變更: The buffered headers are written to the output stream.

flush_headers()

Finally send the headers to the output stream and flush the internal headers buffer.

在 3.3 版被加入.

log_request(*code='-'*, *size='-'*)

Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)

Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()

Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string(*timestamp=None*)

Returns the date and time given by *timestamp* (which must be `None` or in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()

Returns the current date and time, formatted for logging.

address_string()

Returns the client address.

在 3.3 版的變更: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)

This class serves files from the directory *directory* and below, or the current directory if *directory* is not provided, directly mapping the directory structure to HTTP requests.

在 3.7 版的變更: 新增 *directory* 參數。

在 3.9 版的變更: The *directory* parameter accepts a *path-like object*.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version

This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types, contains custom overrides for the default system mappings. The mapping is used case-insensitively, and so should contain only lower-cased keys.

在 3.9 版的變更: This dictionary is no longer filled with the default system mappings, but only contains overrides.

SimpleHTTPRequestHandler 類🄵定義了以下方法:

do_HEAD()

This method serves the `'HEAD'` request type: it sends the headers it would send for the equivalent `GET` request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is

generated by calling the `list_directory()` method. This method uses *os.listdir()* to scan the directory, and returns a `404` error response if the *listdir()* fails.

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a `404`, `'File not found'` error. If there was an `'If-Modified-Since'` header in the request, and the file was not modified after this time, a `304`, `'Not Modified'` response is sent. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output.

For example usage, see the implementation of the `test` function in Lib/http/server.py.

在 3.7 版的變更: Support of the `'If-Modified-Since'` header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic web-server serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

SimpleHTTPRequestHandler can also be subclassed to enhance behavior, such as using different index file names by overriding the class attribute `index_pages`.

class `http.server.`**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

> **備🄵**
>
> CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used --- the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

CGIHTTPRequestHandler 類🄵定義了以下資料成員:

cgi_directories

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

CGIHTTPRequestHandler 類🄵定義了以下方法:

do_POST()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

Deprecated since version 3.13, will be removed in version 3.15: *CGIHTTPRequestHandler* is being removed in 3.15. CGI has not been considered a good way to do things for well over a decade. This code has been unmaintained for a while now and sees very little practical use. Retaining it could lead to further *security considerations*.

22.17.1 命令列介面

http.server can also be invoked directly using the `-m` switch of the interpreter. The following example illustrates how to serve files relative to the current directory:

```
python -m http.server [OPTIONS] [port]
```

可接受以下選項:

port

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

-b, --bind <address>

Specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. By default, the server binds itself to all interfaces. For example, the following command causes the server to bind to localhost only:

```
python -m http.server --bind 127.0.0.1
```

在 3.4 版被加入.

在 3.8 版的變更: 於 `--bind` 選項中支援 IPv6。

-d, --directory <dir>

Specifies a directory to which it should serve the files. By default, the server uses the current directory. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

在 3.7 版被加入.

-p, --protocol <version>

Specifies the HTTP version to which the server is conformant. By default, the server is conformant to HTTP/1.0. For example, the following command runs an HTTP/1.1 conformant server:

```
python -m http.server --protocol HTTP/1.1
```

在 3.11 版被加入.

--cgi

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15: *http.server* command line `--cgi` support is being removed because *CGIHTTPRequestHandler* is being removed.

> **警告**
>
> *CGIHTTPRequestHandler* and the `--cgi` command-line option are not intended for use by untrusted clients and may be vulnerable to exploitation. Always use within a secure environment.

--tls-cert

Specifies a TLS certificate chain for HTTPS connections:

```
python -m http.server --tls-cert fullchain.pem
```

在 3.14 版被加入.

--tls-key

Specifies a private key file for HTTPS connections.

This option requires `--tls-cert` to be specified.

在 3.14 版被加入.

--tls-password-file

Specifies the password file for password-protected private keys:

```
python -m http.server \
       --tls-cert cert.pem \
       --tls-key key.pem \
       --tls-password-file password.txt
```

This option requires *--tls-cert*‘ to be specified.

在 3.14 版被加入.

22.17.2 安全性注意事項

SimpleHTTPRequestHandler will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default *BaseHTTPRequestHandler* `.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

在 3.12 版的變更: Control characters are scrubbed in stderr logs.

22.18 `http.cookies` --- HTTP 狀態管理

原始碼： Lib/http/cookies.py

---

http.cookies 模組定義了 cookie 概念的抽象化的類🄵，cookie 是一種 HTTP 狀態管理機制。它支援簡單的純字串 cookie，也提供了將任何可序列化的資料型🄵作🄵 cookie 值的抽象化。

此模組原先嚴格遵循了 **RFC 2109** 和 **RFC 2068** 規範所描述的剖析規則。自從發現 MSIE 3.0x 🄵不遵循這些規範所描述的字元規則，許多目前的🄵覽器和伺服器也在處理 cookie 時放寬了剖析規則。因此，此模組現在使用的剖析規則比之前的更🄵寬松。

字元集*string.ascii_letters*、*string.digits* 和 `!#$%&'*+-.^_`|~` 表示此模組在 cookie 名稱 (*key*) 中允許的合法字元集合。

在 3.3 版的變更: 允許':' 作🄵有效的 cookie 名稱字元。

備🄵

當遇到無效的 cookie 時，會引發*CookieError*，因此如果你的 cookie 資料來自🄵覽器，在剖析時你應該總是🄵無效資料作准備🄵捕捉*CookieError*。

exception http.cookies.**CookieError**

因🄵不符合 **RFC 2109** 而引發的例外：不正確的屬性、不正確的 *Set-Cookie* 標頭等。

class http.cookies.**BaseCookie**([*input*])

這個類🄵是一個類似字典的物件，其中的鍵是字串，而值則是*Morsel* 實例。請注意，當設定鍵的值時，值會先被轉🄵🄵一個包含該鍵和值的*Morsel*。

如果有給定 *input*，它會被傳遞給*load()* 方法。

class http.cookies.**SimpleCookie**([*input*])

這個類🄵繼承自*BaseCookie* 🄵覆寫了*value_decode()* 和*value_encode()*。SimpleCookie 支援字串作🄵 cookie 值。當設定值時，SimpleCookie 會呼叫🄵建的*str()* 來將值轉🄵🄵字串。從 HTTP 接收的值會保持🄵字串。

也參考

***http.cookiejar* 模組**
用於網路用🄵端的 HTTP cookie 處理。*http.cookiejar* 和*http.cookies* 模組互不相依。

RFC 2109 - HTTP 狀態管理機制
這是此模組實作的狀態管理規範。

22.18.1 Cookie 物件

BaseCookie.**value_decode**(*val*)

從字串表示回傳 (real_value, coded_value) 的元組。real_value 可以是任何型🄵。此方法在*BaseCookie* 中不做解碼 --- 它存在以便可以被覆寫。

BaseCookie.**value_encode**(*val*)

回傳一個元組 (real_value, coded_value)。*val* 可以是任何型🄵，但 coded_value 總是會被轉🄵🄵字串。此方法在*BaseCookie* 中不做編碼 --- 它存在以便可以被覆寫。

一般來🄵，*value_encode()* 和*value_decode()* 應該在 *value_decode* 的範圍🄵是互逆的 (inverse)。

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)

回傳一個適合作🄵 HTTP 標頭來傳送的字串表示。*attrs* 和 *header* 會傳送給每個*Morsel* 的*output()* 方法。*sep* 用來連接標頭，預設🄵 '\r\n' (CRLF) 的組合。

BaseCookie.**js_output**(*attrs=None*)

回傳一個可嵌入的 JavaScript 片段，如果在支援 JavaScript 的🄵覽器上執行，它的行🄵會與 HTTP 標頭被傳送的情🄵相同。

attrs 的意義與*output()* 🄵的相同。

BaseCookie.**load**(*rawdata*)

如果 *rawdata* 是字串，會將其作🄵 HTTP_COOKIE 剖析，🄵將在其中找到的值當作*Morsel* 新增。如果它是一個字典，則等同於：

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.18.2 Morsel 物件

class http.cookies.**Morsel**

抽象化一個鍵 / 值對，它有一些 **RFC 2109** 屬性。

Morsel 是一種類似字典的物件，其鍵的集合是固定的 --- 有效的 **RFC 2109** 屬性，它們是：

expires
path
comment
domain
max-age
secure
version
httponly
samesite
partitioned

屬性*httponly* 指定 cookie 僅在 HTTP 請求中傳輸，而不可通過 JavaScript 存取。這是🄵了🄵輕某些形式的跨網站🄵本攻擊。

samesite 屬性控制🄵覽器要在何時將 cookie 與跨網站請求一起傳送，這有助於🄵輕 CSRF 攻擊。有效值🄵"Strict"（僅在同站請求時傳送）、"Lax"（同站請求和頂層導航時傳送）和"None"（同站請求和跨站請求時傳送）。使用"None" 時，必須同時設定"secure" 屬性，這是現代🄵覽器的要求。

The attribute *partitioned* indicates to user agents that these cross-site cookies *should* only be available in the same top-level context that the cookie was first set in. For this to be accepted by the user agent, you **must** also set `Secure`.

In addition, it is recommended to use the `__Host` prefix when setting partitioned cookies to make them bound to the hostname and not the registrable domain. Read CHIPS (Cookies Having Independent Partitioned State) for full details and examples.

鍵不區分大小寫，其預設值🄵 `''`。

在 3.5 版的變更: `__eq__()` 現在會考慮*key* 和*value*。

在 3.7 版的變更: 屬性*key*、*value* 和*coded_value* 是唯讀的。請使用*set()* 來設定它們。

在 3.8 版的變更: 新增對*samesite* 屬性的支援。

在 3.14 版的變更: 新增對*partitioned* 屬性的支援。

Morsel.**value**

cookie 的值。

Morsel.**coded_value**

cookie 的編碼值 --- 這是應該被傳送的值。

Morsel.**key**

cookie 的名稱。

Morsel.**set**(*key*, *value*, *coded_value*)

設定 *key*、*value* 和 *coded_value* 屬性。

Morsel.**isReservedKey**(*K*)

K 是否🄵*Morsel* 的鍵集合中的成員。

Morsel.**output**(*attrs=None*, *header='Set-Cookie:'*)

回傳適合作🄵 HTTP 標頭傳送的 Morsel 的字串表示。預設會包含所有屬性，除非有給定 *attrs*，在此情🄵下，它應該是一個要使用的屬性的串列。預設的 *header* 是 `"Set-Cookie:"`。

Morsel.**js_output**(*attrs=None*)

回傳一個可嵌入的 JavaScript 片段，如果在支援 JavaScript 的🄴覽器上執行，它的行🄴會與 HTTP 標頭被傳送的情🄴相同。

attrs 的意義與*output()* 🄵的相同。

Morsel.**OutputString**(*attrs=None*)

回傳表示 Morsel 的字串，不包含周圍任何的 HTTP 或 JavaScript。

attrs 的意義與*output()* 🄵的相同。

Morsel.**update**(*values*)

更新 Morsel 字典中的值🄵 *values* 字典中的值。如果 *values* 字典中的任何鍵不是有效的 **RFC 2109** 屬性則引發錯誤。

在 3.5 版的變更: 對於無效的鍵會引發錯誤。

Morsel.**copy**(*value*)

回傳 Morsel 物件的淺層🄵🄵。

在 3.5 版的變更: 回傳 Morsel 物件而不是字典。

Morsel.**setdefault**(*key*, *value=None*)

如果鍵不是一個有效的 **RFC 2109** 屬性會引發錯誤，否則行🄴與*dict.setdefault()* 相同。

22.18.3 范例

以下範例示範如何使用*http.cookies* 模組。

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # 🄵生 HTTP 標頭
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # 同上
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # 從字串 (HTTP 標頭) 載入
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
```

(繼續下一頁)

(繼續上一頁)

```
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # 等同於 C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.19 `http.cookiejar` --- HTTP 用🄵端的 Cookie 處理

原始碼： Lib/http/cookiejar.py

---

http.cookiejar 模組定義了自動處理 HTTP cookie 的類🄵。當需要存取小量資料——*cookies*——的網站時會非常有用。這些資料會依據網頁伺服器的 HTTP 回應來設置於用🄵端機器，🄵在之後的請求中被回傳到伺服器。

一般的 Netscape cookie 協定和 **RFC 2965** 定義的 cookie 協定都會被處理。RFC 2965 的處理預設是關閉的。**RFC 2109** cookie 會被剖析🄵 Netscape cookie，然後根據有生效的「策略」來被看作🄵 Netscape 或 RFC 2965 cookie。*http.cookiejar* 會嘗試遵循實際上的 Netscape cookie 協定（與原始的 Netscape 規格中的協定有很大的不同），包括 RFC 2965 所引進的 `max-age` 和 `port` cookie 屬性。

> **備🄵**
>
> 在 *Set-Cookie* 和 *Set-Cookie2* 標頭中出現的各種命名參數（例如 `domain` 和 `expires`）通常被稱🄵 *attributes*。🄵了與 Python 的屬性分🄵開，這個模組的🄵明檔案改用 *cookie-attribute* 來稱呼。

模組定義了以下例外：

exception http.cookiejar.**LoadError**

當從檔案載入 cookies 失敗時，*FileCookieJar* 的實例會引發這個例外。*LoadError* 是*OSError* 的子類🄵。

在 3.3 版的變更: *LoadError* 以前是*IOError* 的子型🄵，現在是*OSError* 的🄵名。

這個模組提供了以下類🄵：

class http.cookiejar.**CookieJar**(*policy=None*)

policy 是實作了*CookiePolicy* 介面的一個物件。

CookieJar 類🄵會儲存 HTTP cookies。它會從 HTTP 請求中抽取出 cookies，🄵在 HTTP 回應中將它們回傳。*CookieJar* 實例會在必要時自動使包含的 cookies 過期。它的子類🄵也負責從檔案或資料庫中儲存和🄵取 cookies。

class http.cookiejar.**FileCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

policy 是實作了*CookiePolicy* 介面的一個物件。至於其他引數，請參🄵對應屬性的🄵明文件。

一個可以從磁碟上的檔案載入 Cookie，也可以儲存 Cookie 到磁碟上檔案的*CookieJar*。在*load()* 或*revert()* 方法被呼叫之前，Cookie 🄵**不會**從指定的檔案載入。這個類🄵的子類🄵有記載於*FileCookieJar 子類🄵及與網頁🄵覽器的合作* 章節。

這個類🄵不應該直接初始化 - 請使用下面的子類🄵代替。

在 3.8 版的變更: filename 參數支援傳入一個*path-like object*。

class http.cookiejar.**CookiePolicy**

此類🄴負責🄴定是否應從伺服器接受 / 回傳每個 cookie。

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=('https', 'wss')*)

建構函式引數只能以關鍵字引數的形式傳送。*blocked_domains* 是我們從不接受 cookies 或回傳 cookies 的網域名稱序列。*allowed_domains* 如果不是*None*，這是我們接受🄴回傳 cookies 的唯一網域名稱序列。*secure_protocols* 是可以加入安全 cookies 的通訊協定序列。預設情🄴下，*https* 和 *wss*（安全 websocket）會被視🄴安全通訊協定。所有其他引數請參🄴*CookiePolicy* 及*DefaultCookiePolicy* 物件的🄴明文件。

DefaultCookiePolicy 實作了 Netscape 和 **RFC 2965** cookies 的標准接受/拒🄴規則。預設情🄴下，**RFC 2109** cookies（即在 *Set-Cookie* 標頭中接收到的版本 cookie-attribute 🄵 1 的 cookies）會根據 RFC 2965 規則處理。但是，如果 RFC 2965 處理被關閉，或者*rfc2109_as_netscape* 是 `True`，RFC 2109 cookie 會被*CookieJar* 實例「降級」🄵 Netscape cookie，方法是將*Cookie* 實例的 `version` 屬性設🄵 0。*DefaultCookiePolicy* 也提供了一些參數來允許對策略進行一些微調。

class http.cookiejar.**Cookie**

這個類🄴代表 Netscape、**RFC 2109** 和 **RFC 2965** cookies。我們不希望*http.cookiejar* 的使用者建立自己的*Cookie* 實例。如有必要，請在*CookieJar* 實例上呼叫 `make_cookies()`。

也參考

***urllib.request* 模組**
URL 打開時會自動處理 cookie。

***http.cookies* 模組**
HTTP cookie 類🄵，主要用於伺服器端程式碼。*http.cookiejar* 和*http.cookies* 模組不互相依賴。

https://curl.se/rfc/cookie_spec.html
原始 Netscape cookie 協定的規範。雖然這仍是主流協定，但所有主要🄵覽器（以及*http.cookiejar*）實作的「Netscape cookie 協定」與 `cookie_spec.html` 中實作的協定只有一點相似之處。

RFC 2109 - HTTP 狀態管理機制
被 **RFC 2965** 所取代，會使用 *Set-Cookie* 設定 version=1。

RFC 2965 - HTTP 狀態管理機制
有錯誤修正的 Netscape 通訊協定。使用 *Set-Cookie2* 取代 *Set-Cookie*。未被廣泛使用。

https://kristol.org/cookie/errata.html
未完成的 **RFC 2965** 勘誤表。

RFC 2964 - HTTP 狀態管理使用方法

22.19.1 CookieJar 與 FileCookieJar 物件

CookieJar 物件支援*iterator* 協定來遍歷包含的*Cookie* 物件。

CookieJar 擁有以下方法：

CookieJar.**add_cookie_header**(*request*)

將正確的 *Cookie* 標頭加入 *request*。

如果策略允許（即*CookieJar* 的*CookiePolicy* 實例的 rfc2965 和 hide_cookie2 屬性分🄵🄵 true 和 false），*Cookie2* 標頭也會在適當的時候加入。

如*urllib.request* 文件所述，*request* 物件（通常是*urllib.request.Request* 的實例）必須支援 get_full_url()、has_header()、get_header()、header_items()、add_unredirected_header() 方法與 host、type、unverifiable、origin_req_host 屬性。

在 3.3 版的變更: *request* 物件需要 origin_req_host 屬性。已移除對已廢🄵方法 get_origin_req_host() 的依賴。

CookieJar.**extract_cookies**(*response*, *request*)

在策略允許的情🄵下，從 HTTP *response* 抽取出 cookies 🄵儲存到*CookieJar* 中。

CookieJar 會在 *response* 引數中尋找允許的 *Set-Cookie* 和 *Set-Cookie2* 標頭，🄵視情🄵儲存 cookies（需經*CookiePolicy.set_ok()* 方法的檢測）。

response 物件（通常是呼叫*urllib.request.urlopen()* 的結果或類似的東西）應該支援 info() 方法，它會回傳一個*email.message.Message* 的實例。

如*urllib.request* 的檔案所🄵，*request* 物件（通常是*urllib.request.Request* 的實例）必須支援 get_full_url() 方法以及 host、unverifiable 和 origin_req_host 等屬性，該請求被用於設置 cookie-attributes 的預設值🄵檢查 cookie 是否允許被設置。

在 3.3 版的變更: *request* 物件需要 origin_req_host 屬性。已移除對已廢🄵方法 get_origin_req_host() 的依賴。

CookieJar.**set_policy**(*policy*)

設定要使用的*CookiePolicy* 實例。

CookieJar.**make_cookies**(*response*, *request*)

回傳從 *response* 物件抽取出的*Cookie* 物件序列。

有關 *response* 和 *request* 引數所需的介面，請參🄵*extract_cookies()* 的🄵明檔案。

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)

如果策略允許這🄵做，設定*Cookie*。

CookieJar.**set_cookie**(*cookie*)

設定一個*Cookie*，不經過策略檢查。

CookieJar.**clear**(*[domain[, path[, name]]]*)

清除一些 cookies。

如果不帶引數地叫用，則清除所有 cookies。如果給定單一引數，則只移除屬於該 *domain* 的 cookies。如果給定兩個引數，則會移除屬於指定 *domain* 和 URL *path* 的 cookie。如果給定三個引數，則會移除指定 *domain*、*path* 和 *name* 的 cookie。

如果🄵有符合的 cookie 存在，則引發*KeyError*。

CookieJar.**clear_session_cookies**()

🄵除所有 session cookies。

🄵除所有包含有 true discard 屬性的 cookie (通常是因🄵它們🄵有 *max-age* 或 *expires* cookie 屬性，或有明確的 *discard* cookie 屬性)。對於互動式🄵覽器，工作階段的結束通常對應於關閉🄵覽器視窗。

請注意 save() 方法無論如何都不會儲存 session cookies，除非你透過傳入 true *ignore_discard* 引數來要求。

FileCookieJar 實例有下列附加方法：

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

將 cookies 儲存到檔案中。

這個基底類🄵會🄵生*NotImplementedError*。子類🄵可以不實作此方法。

filename 是儲存 cookies 的檔案名稱。如果 *filename* 未指定，則會使用 `self.filename`（其預設值是傳給建構函式的值（如果有））；如果 `self.filename` 是*None*，則會🄵生*ValueError*。

ignore_discard: 即使設定🄵舍🄵的 cookies 也會儲存。*ignore_expires*: 保存過期的 cookie

如果檔案已經存在，則會被覆寫，因此會抹除其中包含的所有 cookies。儲存的 cookies 可以稍後使用*load()* 或*revert()* 方法還原。

`FileCookieJar.`**`load`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

從檔案中載入 cookies。

舊的 cookies 會被保留，除非被新載入的 cookies 覆蓋。

引數與*save()* 相同。

被命名的檔案必須是類🄵所能正確剖析的格式，否則會🄵生*LoadError*。此外，*OSError* 可能會被引發，例如檔案不存在時。

在 3.3 版的變更: 以前會引發*IOError*，現在它是*OSError* 的🄵名。

`FileCookieJar.`**`revert`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

清除所有 cookies，🄵從儲存的檔案重新載入 cookies。

revert() 可以🄵生與*load()* 相同的例外。如果失敗，物件的狀態不會被改變。

FileCookieJar 實例擁有以下公開屬性:

`FileCookieJar.`**`filename`**

保存 cookie 的預設檔案的檔案名稱。此屬性可以被指定。

`FileCookieJar.`**`delayload`**

若🄵 true，則從硬盤惰性地載入 cookies。此屬性不應被指定。這只是一個提示，因🄵這只會影響效能，不會影響行🄵 (除非磁碟上的 cookies 正在改變)。一個*CookieJar* 物件可以忽略它。🄵有一個包含在標準函式庫中的*FileCookieJar* 類🄵會惰性地載入 cookies。

22.19.2 FileCookieJar 子類🄵及與網頁🄵覽器的合作

以下*CookieJar* 子類🄵提供用於讀取和寫入。

`class` `http.cookiejar.`**`MozillaCookieJar`**(*filename=None*, *delayload=None*, *policy=None*)

一個能🄵以 Mozilla `cookies.txt` 檔案格式（該格式也被 curl 和 Lynx 以及 Netscape 🄵覽器所使用）從磁碟載入和儲存 cookie 的*FileCookieJar*。

> **備🄵**
>
> 這會遺失關於 **RFC 2965** cookie 的訊息，以及較新或非標準 cookie 屬性（例如 `port`）的訊息。

> **警告**
>
> 如果 Cookie 遺失或損壞會造成不便，請在儲存之前先備份 Cookie（有些微妙的情🄵可能會導致檔案在載入/儲存往返過程中發生輕微變化）。

另外請注意，在 Mozilla 執行時儲存的 cookies 會被 Mozilla 🄵除。

`class` `http.cookiejar.`**`LWPCookieJar`**(*filename=None*, *delayload=None*, *policy=None*)

一個*FileCookieJar* 可以從 libwww-perl 函式庫的 *Set-Cookie3* 檔案格式載入 cookies 🄵儲存 cookies 到磁碟。如果你想要將 cookie 儲存在人類可讀的檔案中，這是很方便的。

在 3.8 版的變更: filename 參數支援傳入一個*path-like object*。

22.19.3 CookiePolicy 物件

實作*CookiePolicy* 介面的物件有以下方法：

CookiePolicy.**set_ok**(*cookie*, *request*)

回傳值表示的是否接受伺服器 cookie 的布林值。

cookie 是一個*Cookie* 實例。*request* 是一個實作了*CookieJar.extract_cookies()* 文件所定義的介面的物件。

CookiePolicy.**return_ok**(*cookie*, *request*)

回傳布林值，表示 cookie 是否應回傳伺服器。

cookie 是一個*Cookie* 實例。*request* 是一個實作了*CookieJar.add_cookie_header()* 文件所定義的介面的物件。

CookiePolicy.**domain_return_ok**(*domain*, *request*)

在給定 cookie 網域如果不應回傳 cookie，則回傳 `False`。

此方法🄵一種最佳化。它消除了檢查每個具有特定網域的 cookie 的需要（這可能需要讀取許多檔案）。從*domain_return_ok()* 和*path_return_ok()* 回傳 true 會把所有的工作留給*return_ok()*。

如果*domain_return_ok()* 對 cookie 網域回傳 true，*path_return_ok()* 就會被用來呼叫 cookie 路徑。否則*path_return_ok()* 和*return_ok()* 永遠不會被呼叫。如果*path_return_ok()* 回傳 true，則會呼叫*return_ok()* 附帶的*Cookie* 物件本身以進行全面檢查。否則，*return_ok()* 將永遠不會被呼叫。

請注意，*domain_return_ok()* 是針對每個 *cookie* 網域來呼叫的，而不只是針對 *request* 網域。例如，如果請求網域是 `"www.example.com"`，這個函式可能會同時被 `".example.com"` 和 `"www.example.com"` 呼叫。同樣的道理也適用於*path_return_ok()*。

request 引數與*return_ok()* 的🄵明文件相同。

CookiePolicy.**path_return_ok**(*path*, *request*)

給定 cookie 路徑如不應該回傳 cookie，則回傳 `False` 。

關於*domain_return_ok()* 請見文件。

除了實作上述方法外，*CookiePolicy* 介面的實作也必須提供下列屬性，指出應該使用哪些通訊協定，以及如何使用。所有這些屬性都可以被指定。

CookiePolicy.**netscape**

實作 Netscape 協定。

CookiePolicy.**rfc2965**

實作 **RFC 2965** 協定。

CookiePolicy.**hide_cookie2**

不要在請求中加入 *Cookie2* 標頭（這個標頭的存在向伺服器表明我們能🄵正確剖析 **RFC 2965** cookie）。

以*DefaultCookiePolicy* 🄵子類🄵定義*CookiePolicy* 類🄵，🄵覆寫其中的一些或全部方法是最有用的。*CookiePolicy* 本身可以用做’無策略’，允許設定和接收任何和所有的 cookies (這不太可能有用)。

22.19.4 DefaultCookiePolicy 物件

執行接受和回傳 cookie 的標准規則。

同時支援 **RFC 2965** 和 Netscape cookies。RFC 2965 的處理預設🄵關閉。

提供你自己的策略的最簡單方法是覆寫此類🄵，🄵在你覆寫的實作中呼叫其方法，然後再加入你自己的額外檢查：

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

除了實作*CookiePolicy* 介面所需的功能外，這個類🄵還允許你阻止或允許網域名稱設定和接收 cookies。還有調整嚴格度的控制選項，可讓你稍微收緊相對寬松的 Netscape 通訊協定（代價是阻擋一些良性 cookies）。

提供網域阻止清單和允許清單（預設都關閉）。只有不在阻止清單中和在允許清單中（如果允許清單是有效的）的網域才參與 cookie 設定和回傳。使用 *blocked_domains* 構造函式引數，以及 `blocked_domains()` 和 `set_blocked_domains()` 方法（以及 *allowed_domains* 的相應引數和方法）。如果你設定了允許清單，你可以透過設定🄵*None* 來再次關閉它。

封鎖或允許清單中不以點開頭的網域必須與 cookie 網域相等才能匹配。例如，`"example.com"` 會與封鎖清單項目 `"example.com"` 匹配，但 `"www.example.com"` 則不會。以點開頭的網域也會與更特定的網域相匹配。例如，`"www.example.com"` 和 `"www.coyote.example.com"` 都匹配 `".example.com"`（但 `"example.com"` 本身不匹配）。IP 位址是例外，它們必須完全匹配。例如，如果 blocked_domains 包含 `"192.168.1.2"` 和 `".168.1.2"`，則 192.168.1.2 會被封鎖，但 193.168.1.2 則不會。

DefaultCookiePolicy 實作了下列附加方法：

`DefaultCookiePolicy.`**`blocked_domains`**`()`
: 回傳被阻止網域的序列（以元組形式）。

`DefaultCookiePolicy.`**`set_blocked_domains`**`(`*blocked_domains*`)`
: 設定阻止網域的順序。

`DefaultCookiePolicy.`**`is_blocked`**`(`*domain*`)`
: 如果 *domain* 在設定或接收 cookies 的阻止列表上，則回傳 `True`。

`DefaultCookiePolicy.`**`allowed_domains`**`()`
: 回傳*None*，或允許網域的序列（以元組形式）。

`DefaultCookiePolicy.`**`set_allowed_domains`**`(`*allowed_domains*`)`
: 設定允許網域的序列，或*None*。

`DefaultCookiePolicy.`**`is_not_allowed`**`(`*domain*`)`
: 如果 *domain* 不在設定或接收 cookies 的允許清單上，則回傳 `True`。

DefaultCookiePolicy 實例有以下屬性，這些屬性都是由同名的建構函式引數初始化，而且都可以被指定。

`DefaultCookiePolicy.`**`rfc2109_as_netscape`**
: 如果🄵 true，請求*CookieJar* 實例透過設定*Cookie* 實例的版本屬性🄵 0，將 **RFC 2109** cookies (即在 *Set-Cookie* 標頭中收到的、版本 cookie 屬性🄵 1 的 cookies) 降級🄵 Netscape cookies。預設值是*None*，在這種情🄵下，如果且僅如果 **RFC 2965** 處理被關閉，RFC 2109 cookies 才會被降級。因此，RFC 2109 cookie 在預設情🄵下被降級。

一般的調整嚴格度的控制選項：

`DefaultCookiePolicy.`**`strict_domain`**
: 不允許網站設定具有國家代碼頂層域名的兩層域名，如 `.co.uk`、`.gov.uk`、`.co.nz` 等。這遠遠不🄵完美，也不能保證一定有效！

RFC 2965 通訊協定調整嚴格度的控制選項：

`DefaultCookiePolicy.`**`strict_rfc2965_unverifiable`**
: 遵循針對不可驗證事務（transaction）的 **RFC 2965** 規則（不可驗證事務通常是由重定向或請求發🄵在其它網站的圖片導致的）。如果該屬性🄵假值，則永遠不會基於可驗證性而阻止 cookie

Netscape 通訊協定調整嚴格度的控制選項:

DefaultCookiePolicy.**strict_ns_unverifiable**

即便是對 Netscape cookie 也要應用 **RFC 2965** 規則。

DefaultCookiePolicy.**strict_ns_domain**

指明針對 Netscape cookie 的網域匹配規則的嚴格程度。可接受的值見下文。

DefaultCookiePolicy.**strict_ns_set_initial_dollar**

忽略 Set-Cookie: 標頭中名稱以 `'$'` 開頭的 cookies。

DefaultCookiePolicy.**strict_ns_set_path**

不允許設置路徑與請求 URL 路徑不匹配的 cookie。

strict_ns_domain 是一系列旗標。它的值由將旗標按位元或組成（例如，DomainStrictNoDots|DomainStrictNonDomain 表示兩個旗標都被設定）。

DefaultCookiePolicy.**DomainStrictNoDots**

設定 cookie 時，「host 前綴」不得包含點（例如，www.foo.bar.com 不能🄵 .bar.com 設定 cookie，因🄵 www.foo 包含一個點）。

DefaultCookiePolicy.**DomainStrictNonDomain**

未明確指定 domain cookie-attribute 的 Cookie 只能回傳給與設定 Cookie 的網域相同的網域（例如，spam.example.com 將不會回傳來自 example.com 但🄵有 domain cookie-attribute 的 Cookie）。

DefaultCookiePolicy.**DomainRFC2965Match**

設定 cookie 時，需要完整的 **RFC 2965** 網域匹配。

🄵了方便起見，以下屬性是上述旗標最有用的組合:

DefaultCookiePolicy.**DomainLiberal**

等於 0（即關閉上述所有 Netscape 網域嚴格度旗標）。

DefaultCookiePolicy.**DomainStrict**

等價於 DomainStrictNoDots|DomainStrictNonDomain。

22.19.5 Cookie 物件

Cookie 實例的 Python 屬性大致對應於各種 cookie 標準中指定的標準 cookie 屬性。這個對應關🄵不是一一對應的，因🄵有🄵雜的規則來指定預設值，因🄵 max-age 和 expires cookie 屬性包含相等的訊息，也因🄵 **RFC 2109** cookie 可能會被*http.cookiejar* 從版本 1「降級」到版本 0 (Netscape) cookie。

除了在*CookiePolicy* 方法中的罕見情🄵外，不應該對這些屬性進行指定。這個類🄵不會🄵制執行🄵部一致性，所以如果你這🄵做，你應該要知道你在做什🄵。

Cookie.**version**

整數或*None*。Netscape cookie 的*version* 🄵 0。**RFC 2965** 和 **RFC 2109** cookie 的 version cookie 屬性🄵 1。但是，請注意*http.cookiejar* 可能會將 RFC 2109 cookie 「降級」🄵 Netscape cookie，在這種情🄵下*version* 🄵 0。

Cookie.**name**

Cookie 名稱（字串）。

Cookie.**value**

Cookie 值（字串），或*None*。

Cookie.**port**

代表一個連接埠或一組連接埠的字串 (例如「80」，或「880,8080」)，或*None*。

Cookie.**domain**

Cookie 網域（字串）。

Cookie.**path**
Cookie 路徑（字串，例如 '/acme/rocket_launchers'）。

Cookie.**secure**
如果 cookie 只能透過安全連🄵回傳，則🄵 True。

Cookie.**expires**
自 epoch 起計算的整數到期時間，以秒🄵單位，或*None*。另請參🄵*is_expired()* 方法。

Cookie.**discard**
如果這是 session cookie，即🄵 True。

Cookie.**comment**
來自伺服器解釋此 cookie 功能的字串🄵解，或*None*。

Cookie.**comment_url**
鏈接到來自伺服器的解釋此 cookie 功能的🄵釋的 URL，或者🄵*None*。

Cookie.**rfc2109**
True 如果這個 cookie 是以 **RFC 2109** cookie 的形式收到的 (即這個 cookie 是在 *Set-Cookie* 標頭中收到的，而且在那個標頭中的 Version cookie-attribute 的值是 1)。提供這個屬性的原因是*http.cookiejar* 可能會將 RFC 2109 cookie 「降級」🄵 Netscape cookie，在這種情🄵下*version* 的值是 0。

Cookie.**port_specified**
如果伺服器顯式地指定了一個連接埠或一組連接埠（在 *Set-Cookie* / *Set-Cookie2* 標頭中)。

Cookie.**domain_specified**
如果伺服器明確指定網域，則🄵 True。

Cookie.**domain_initial_dot**
如果伺服器明確指定的網域以點 ('.') 開頭，則🄵 True。

Cookies 可能具有額外的非標准 cookie 屬性。你可以使用下列方法存取這些屬性:

Cookie.**has_nonstandard_attr**(*name*)
如果 cookie 具有指定的 cookie 屬性，則回傳 True 。

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
如果 cookie 具有指定的 cookie 屬性，則回傳其值。否則，回傳 *default*。

Cookie.**set_nonstandard_attr**(*name*, *value*)
設定指定 cookie 屬性的值。

Cookie 類🄵也定義了以下方法:

Cookie.**is_expired**(*now=None*)
如果 cookie 已經超過伺服器要求的過期時間，則回傳 True 。如果給定了 *now* （以自紀元起的秒數🄵單位)，則會回傳 cookie 是否已在指定時間過期。

22.19.6 范例

第一個範例展示了*http.cookiejar*: 最常見的用法:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

本範例🄵明如何使用你的 Netscape、Mozilla 或 Lynx cookies 打開 URL (假設 Unix/Netscape 會慣用 cookies 檔案的位置):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

下一個範例🄵明*DefaultCookiePolicy* 的使用。開🄵 **RFC 2965** cookies，在設定和回傳 Netscape cookies 時對網域更嚴格，🄵阻止某些網域設定 cookies 或讓其回傳：

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.20 `xmlrpc` --- XMLRPC 伺服器與用🄵模組

XML-RPC 是一種遠端程序呼叫 (Remote Procedure Call) 方法，它使用通過 HTTP 傳輸（transport）的 XML 來做傳遞。有了它，用🄵端可以在遠端伺服器上呼叫帶有參數的方法（伺服器以 URI 命名）🄵取得結構化的資料。

`xmlrpc` 是一個集合了 XML-RPC 伺服器與用🄵端模組實作的套件。這些模組是：

- *xmlrpc.client*
- *xmlrpc.server*

22.21 `xmlrpc.client` --- XML-RPC 用🄵端存取

原始碼： Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

> **⚠ 警告**
>
> The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see *XML 安全性*.

在 3.5 版的變更: For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport`

instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. `[('Header-Name', 'value')]`). If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

在 3.3 版的變更: 新增 *use_builtin_types* 旗標。

在 3.8 版的變更: 新增 *headers* 參數。

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	*bool*
`int`、`i1`、`i2`、`i4`、`i8` 或 `biginteger`	*int* in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` 或 `float`	*float*. Values get the `<double>` tag.
`string`	*str*
`array`	*list* or *tuple* containing conformable elements. Arrays are returned as *lists*.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their `__dict__` attribute is transmitted.
`dateTime.iso8601`	*DateTime* or *datetime.datetime*. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	*Binary*, *bytes* or *bytearray*. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	*decimal.Decimal*. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return);

failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

在 3.5 版的變更: 加入 *context* 引數。

在 3.6 版的變更: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See https://ws.apache.org/xmlrpc/types.html for a description.

也參考

XML-RPC HOWTO
A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection
Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification
The official specification.

22.21.1 ServerProxy 物件

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.`**`listMethods`**`()`

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

在 3.5 版的變更: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.21.2 日期時間物件

class xmlrpc.client.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through `rich comparison` and `__repr__()` methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# 將 ISO8601 字串轉🄴成 datetime 物件
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.21.3 Binary Objects

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.21.4 Fault Objects

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

An int indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.21.5 ProtocolError 物件

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.21.6 MultiCall 物件

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.21.7 便捷的函式

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the *Fault* exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's *None* value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

[1] This approach has been first presented in a discussion on xmlrpc.com.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a *Fault* exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

在 3.3 版的變更: 新增 *use_builtin_types* 旗標。

22.21.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

22.21.9 Example of Client and Server Usage

See *SimpleXMLRPCServer 範例*.

22.22 `xmlrpc.server` --- 基本 XML-RPC 伺服器

原始碼： Lib/xmlrpc/server.py

The *xmlrpc.server* module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *SimpleXMLRPCServer*, or embedded in a CGI environment, using *CGIXMLRPCRequestHandler*.

> **⚠ 警告**
>
> The *xmlrpc.server* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see *XML 安全性*.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to *SimpleXMLRPCRequestHandler*. The *addr* and *requestHandler* parameters are passed to the *socketserver.TCPServer* constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to *xmlrpc.client* and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the *loads()* function and controls which types are processed when date/times values or binary data are received; it defaults to false.

在 3.3 版的變更: The *use_builtin_types* flag was added.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to *xmlrpc.client* and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the *loads()* function and controls which types are processed when date/times values or binary data are received; it defaults to false.

在 3.3 版的變更: The *use_builtin_types* flag was added.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the *SimpleXMLRPCServer* constructor parameter is honored.

22.22.1 SimpleXMLRPCServer 物件

The *SimpleXMLRPCServer* class is based on *socketserver.TCPServer* and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

在 3.7 版的變更: *register_function()* 也可被當作裝飾器使用。

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using *register_function()*. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from

`_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **⚠ 警告**
>
> Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

`SimpleXMLRPCServer.`**`register_introspection_functions`**`()`

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`SimpleXMLRPCServer.`**`register_multicall_functions`**`()`

Registers the XML-RPC multicall function system.multicall.

`SimpleXMLRPCRequestHandler.`**`rpc_paths`**

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer 范例

Server code:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

⚠ 警告

Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
```

(繼續下一頁)

(繼續上一頁)

```
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py`:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```

22.22.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

在 3.7 版的變更: *register_function()* 也可被當作裝飾器使用。

CGIXMLRPCRequestHandler.**register_instance**(*instance*)

Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for

individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.**register_introspection_functions**()

Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

CGIXMLRPCRequestHandler.**register_multicall_functions**()

Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)

Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

範例:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.22.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer*; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

在 3.3 版的變更: The *use_builtin_types* flag was added.

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.**DocXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

22.22.4 DocXMLRPCServer 物件

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.22.5 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.23 ipaddress --- IPv4/IPv6 操作函式庫

原始碼： Lib/ipaddress.py

---

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see ipaddress-howto.

在 3.3 版被加入.

22.23.1 Convenience factory functions

The *ipaddress* module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.**ip_address**(*address*)

Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)

Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than `2**32` will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network*

constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)

Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

22.23.2 IP Addresses

Address objects

The *IPv4Address* and *IPv6Address* objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by *IPv4Address* objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Address**(*address*)

Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0--255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are not tolerated to prevent confusion with octal notation.
2. An integer that fits into 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

在 3.8 版的變更: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

在 3.9.5 版的變更: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc *inet_pton()*.

version

The appropriate version number: `4` for IPv4, `6` for IPv6.

在 3.14 版的變更: Made available on the class.

max_prefixlen

The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

在 3.14 版的變更: Made available on the class.

compressed

exploded

The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed

The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer

The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

在 3.5 版被加入.

is_multicast

`True` if the address is reserved for multicast use. See **RFC 3171** (for IPv4) or **RFC 2373** (for IPv6).

is_private

`True` if the address is defined as not globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exceptions:

- `is_private` is `False` for the shared address space (`100.64.0.0/10`)
- For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *`IPv6Address.ipv4_mapped`*):

  ```
  address.is_private == address.ipv4_mapped.is_private
  ```

`is_private` has value opposite to *`is_global`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

在 3.13 版的變更: Fixed some false positives and false negatives.

- `192.0.0.0/24` is considered private with the exception of `192.0.0.9/32` and `192.0.0.10/32` (previously: only the `192.0.0.0/29` sub-range was considered private).
- `64:ff9b:1::/48` is considered private.
- `2002::/16` is considered private.
- There are exceptions within `2001::/23` (otherwise considered private): `2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`. The exceptions are not considered private.

is_global

`True` if the address is defined as globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exception:

For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *`IPv6Address.ipv4_mapped`*):

```
address.is_global == address.ipv4_mapped.is_global
```

`is_global` has value opposite to *`is_private`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

在 3.4 版被加入.

在 3.13 版的變更: Fixed some false positives and false negatives, see *`is_private`* for details.

is_unspecified

`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved

`True` if the address is noted as reserved by the IETF. For IPv4, this is only `240.0.0.0/4`, the `Reserved` address block. For IPv6, this is all addresses allocated as `Reserved by IETF` for future use.

> **備🄵**
>
> For IPv4, `is_reserved` is not related to the address block value of the `Reserved-by-Protocol` column in iana-ipv4-special-registry.

> **警示**
>
> For IPv6, `fec0::/10` a former Site-Local scoped address prefix is currently excluded from that list (see *`is_site_local`* & **RFC 3879**).

is_loopback

`True` if this is a loopback address. See **RFC 3330** (for IPv4) or **RFC 2373** (for IPv6).

is_link_local

`True` if the address is reserved for link-local usage. See **RFC 3927**.

ipv6_mapped

`IPv4Address` object representing the IPv4-mapped IPv6 address. See **RFC 4291**.

在 3.13 版被加入.

`IPv4Address.`**`__format__`**(*fmt*)

Returns a string representation of the IP address, controlled by an explicit format string. *fmt* can be one of the following: `'s'`, the default option, equivalent to *`str()`*, `'b'` for a zero-padded binary string, `'X'` or `'x'` for an uppercase or lowercase hexadecimal representation, or `'n'`, which is equivalent to `'b'` for IPv4 addresses and `'x'` for IPv6. For binary and hexadecimal representations, the form specifier `'#'` and the grouping option `'_'` are available. `__format__` is used by `format`, `str.format` and f-strings.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

在 3.9 版被加入.

class `ipaddress.`**`IPv6Address`**(*address*)

Construct an IPv6 address. An *`AddressValueError`* is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.

   Optionally, the string may also have a scope zone ID, expressed with a suffix `%scope_id`. If present, the scope ID must be non-empty, and may not contain `%`. See **RFC 4007** for details. For example, `fe80::1234%1` might identify address `fe80::1234` on the first link of the node.

2. An integer that fits into 128 bits.

3. An integer packed into a *bytes* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes and methods, see the corresponding documentation of the *IPv4Address* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

在 3.4 版被加入.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *is_private* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

ipv4_mapped

For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

scope_id

For scoped addresses as defined by **RFC 4007**, this property identifies the particular zone of the address's scope that the address belongs to, as a string. When no scope zone is specified, this property will be `None`.

sixtofour

For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo

For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

`IPv6Address.__format__`(*fmt*)

Refer to the corresponding method documentation in *IPv4Address*.

在 3.9 版被加入.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *str()* and *int()* builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Note that IPv6 scoped addresses are converted to integers without scope zone ID.

運算子

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

比較運算子

Address objects can be compared with the usual set of comparison operators. Same IPv6 addresses with different scope zone IDs are not equal. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

算術運算子

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

22.23.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing *IPv4Address* object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

在 3.5 版的變更: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask

The host mask, as an *IPv4Address* object.

netmask

The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

The total number of addresses in the network.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises *`ValueError`* if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

Return `True` if this network is a subnet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

在 3.7 版被加入.

supernet_of(*other*)

Return `True` if this network is a supernet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

在 3.7 版被加入.

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

在 3.7 版之後被🄵用: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.**IPv6Network**(*address*, *strict=True*)

Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` is not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing *IPv6Address* object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

在 3.5 版的變更: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local

This attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

運算子

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

🄵代

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.23.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of `IPv4Network`, except that arbitrary host addresses are always accepted.

`IPv4Interface` is a subclass of `IPv4Address`, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (`IPv4Address`) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask

Refer to the corresponding attribute documentation in *IPv4Interface*.

運算子

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

22.23.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.**v4_int_to_packed**(*address*)

Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)

Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)

Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.130/32
↪')]
```

ipaddress.**collapse_addresses**(*addresses*)

Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an *iterable* of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

22.23.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.**AddressValueError**(*ValueError*)

Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Any value error related to the net mask.

CHAPTER 23

多媒體服務

此章節所描述的模組 (module) 實作了多種在多媒體服務中相當有用的演算法和介面，🄵可在安裝時🄵定是否要使用它們。以下🄵綜述：

23.1 wave --- 讀寫 WAV 檔案

原始碼： Lib/wave.py

---

wave 模組🄵波形音訊檔案格式「WAVE」（或稱「WAV」）提供了便捷的介面。僅支援未壓縮的 PCM 編碼波形檔。

在 3.12 版的變更: 增加了標頭 WAVE_FORMAT_EXTENSIBLE 的支援，要求的擴展格式🄵KSDATAFORMAT_SUBTYPE_PCM。

wave 模組定義了以下的函式和例外：

wave.**open**(*file*, *mode=None*)

如果 *file* 是一個字串，會打開對應名稱的檔案，否則會以類檔案物件處理。*mode* 可以是：

'rb'
唯讀模式。

'wb'
唯寫模式。

請注意，不支援同時讀寫 WAV 檔案。

mode 設定🄵 'rb' 時，會回傳一個*Wave_read* 物件，*mode* 設定🄵 'wb' 時，則回傳一個*Wave_write* 物件。如果省略 *mode*，🄵且將類檔案 (file-like) 物件作🄵 *file* 參數傳遞，則 file.mode 會是 *mode* 的預設值。

如果你傳遞一個類檔案物件，當呼叫其 close() 方法時，wave 物件不會自動關閉該物件；關閉檔案物件的責任會在呼叫者上。

open() 函式可以在 with 陳述式中使用。當 with 區塊完成時，會呼叫*Wave_read.close()* 或是*Wave_write.close()* 方法。

在 3.4 版的變更: 增加對不可搜尋 (unseekable) 檔案的支援。

exception wave.**Error**

當不符合 WAV 格式或無法操作時會引發錯誤。

23.1.1 Wave_read 物件

class wave.**Wave_read**

讀取一個 WAV 檔案。

由*open()* 回傳的 Wave_read 物件具有以下方法：

close()

關閉*wave* 開🄵的串流🄵使該實例無法使用。當物件回收時自動呼叫。

getnchannels()

回傳音訊聲道的數量（單聲道🄵 `1`，立體聲🄵 `2`）。

getsampwidth()

回傳以位元組表示的取樣寬度 (sample width)。

getframerate()

回傳取樣率。

getnframes()

回傳音訊幀數。

getcomptype()

回傳壓縮類型（僅支援 `'NONE'` 類型）。

getcompname()

getcomptype() 的人類可讀的版本。通常使用 `'not compressed'` 代替 `'NONE'`。

getparams()

回傳一個*namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`，等同於 `get*()` 方法的輸出。

readframes(*n*)

讀取🄵回傳以*bytes* 物件表示的最多 *n* 個音訊幀。

rewind()

重置檔案指標至音訊流的開頭。

以下兩個方法是🄵了與舊的 `aifc` 模組保持相容性，🄵不執行任何操作。

getmarkers()

回傳 `None`。

Deprecated since version 3.13, will be removed in version 3.15: 該方法是🄵了保持與 `aifc` 模組相容性而留存，此模組已於 Python 3.13 中🄵除。

getmark(*id*)

引發錯誤。

Deprecated since version 3.13, will be removed in version 3.15: 該方法是🄵了保持與 `aifc` 模組相容性而留存，此模組已於 Python 3.13 中🄵除。

以下兩個方法所定義的「位置」，在它們之間是相容的，但其他情🄵下則取🄵於具體實作方式。

setpos(*pos*)

將檔案指標設定🄵指定的位置。

tell()

回傳目前的檔案指標位置。

23.1.2 Wave_write 物件

class wave.**Wave_write**

寫入一個 WAV 檔案。

Wave_write 物件，由*open()* 回傳。

對於可搜尋 (seekable) 的輸出串流，wave 標頭將自動更新，以反映實際寫入的幀數。對於不可搜尋的串流，當寫入第一個幀資料時，*nframes* 的值必須是准確的。要取得准確的 *nframes* 值，可以通過呼叫*setnframes()* 或*setparams()* 方法，🄵在呼叫*close()* 之前設定將寫入的幀數量，然後使用*writeframesraw()* 方法寫入幀資料；或者通過呼叫*writeframes()* 方法一次性寫入所有的幀資料。在後一種情🄵下，*writeframes()* 方法將計算資料中的幀數量，🄵在寫入幀資料之前相應地設定 *nframes* 的值。

在 3.4 版的變更: 增加對不可搜尋 (unseekable) 檔案的支援。

Wave_write 物件具有以下方法：

close()

確保 *nframes* 正確，如果該檔案是由*wave* 開🄵的，則關閉該檔案。此方法在物件回收時被呼叫。如果輸出串流不可搜尋且 *nframes* 不符合實際寫入的幀數，則會引發例外。

setnchannels(*n*)

設定音訊的通道數量。

setsampwidth(*n*)

設定取樣寬度🄵 *n* 個位元組。

setframerate(*n*)

設定取樣頻率🄵 *n*。

在 3.2 版的變更: 此方法的非整數輸入會被將四舍五入到最接近的整數。

setnframes(*n*)

設定幀數🄵 *n*。如果實際寫入的幀數不同，則稍後將進行更改（如果輸出串流不可搜尋，則此嘗試將引發錯誤）。

setcomptype(*type*, *name*)

設定壓縮類型和描述。目前只支援壓縮類型🄵 NONE，表示無壓縮。

setparams(*tuple*)

這個 *tuple* 應該是 (nchannels, sampwidth, framerate, nframes, comptype, compname)，值需要是符合 set*() 方法的參數。設定所有參數。

tell()

回傳檔案中的指標位置，其指標位置含意與*Wave_read.tell()* 和*Wave_read.setpos()* 是一致的。

writeframesraw(*data*)

寫入音訊幀，不修正 *nframes*。

在 3.4 版的變更: 現在可接受任何*bytes-like object*。

writeframes(*data*)

寫入音訊幀🄵確保 *nframes* 正確。如果輸出串流不可搜尋，🄵且在寫入 *data* 後已寫入的總幀數與先前設定的 *nframes* 值不符，則會引發錯誤。

在 3.4 版的變更: 現在可接受任何*bytes-like object*。

注意在呼叫*writeframes()* 或*writeframesraw()* 之後設置任何參數都是無效的，任何嘗試這樣做的操作都會引發*wave.Error*。

23.2 `colorsys` --- 🄵色系統間的轉🄵

原始碼： Lib/colorsys.py

`colorsys` 模組 (module) 定義了電腦顯示器所用的 RGB （紅🄵藍）色彩空間與三種其他色彩座標系統：YIQ、HLS (色相、亮度、飽和度) 和 HSV (色相、飽和度、明度) 所表示的🄵色值之間的雙向轉🄵。所有這些色彩空間的座標都使用浮點數值 (floating-point) 來表示。在 YIQ 空間中，Y 座標值🄵 0 和 1 之間，而 I 和 Q 座標均可以🄵正數或負數。在所有其他空間中，座標值均🄵 0 和 1 之間。

也參考

有關色彩空間的更多資訊請見 https://poynton.ca/ColorFAQ.html 和 https://www.cambridgeincolour.com/tutorials/color-spaces.htm。

`colorsys` 模組定義了以下函式：

`colorsys.`**`rgb_to_yiq`**(*r*, *g*, *b*)
: 將🄵色自 RGB 座標轉🄵至 YIQ 座標。

`colorsys.`**`yiq_to_rgb`**(*y*, *i*, *q*)
: 將🄵色自 YIQ 座標轉🄵至 RGB 座標。

`colorsys.`**`rgb_to_hls`**(*r*, *g*, *b*)
: 將🄵色自 RGB 座標轉🄵至 HLS 座標。

`colorsys.`**`hls_to_rgb`**(*h*, *l*, *s*)
: 將🄵色自 HLS 座標轉🄵至 RGB 座標。

`colorsys.`**`rgb_to_hsv`**(*r*, *g*, *b*)
: 將🄵色自 RGB 座標轉🄵至 HSV 座標。

`colorsys.`**`hsv_to_rgb`**(*h*, *s*, *v*)
: 將🄵色自 HSV 座標轉🄵至 RGB 座標。

範例：

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPTER 24

國際化

本章所描述的模組透過提供用於程式訊息中語言的選擇機制或是調整輸出以符合當地慣例，來幫助你編寫不依賴語言和地區設定的軟體

本章節所描述的模組列表🄵:

24.1 gettext --- 多語言國際化服務

原始碼： Lib/gettext.py

The *gettext* module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU **gettext** message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

24.1.1 GNU gettext API

The *gettext* module defines the following API, which is very similar to the GNU **gettext** API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

gettext.**bindtextdomain**(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): *localedir*`/`*language*`/LC_MESSAGES/`*domain*`.mo`, where *language* is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

If *localedir* is omitted or `None`, then the current binding for *domain* is returned.[1]

[1] The default locale directory is system dependent; for example, on Red Hat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults; instead its default is *sys.base_prefix*`/share/locale` (see *sys.base_prefix*). For this reason, it is always best to call *bindtextdomain()* with an explicit absolute path at the start of your application.

gettext.**textdomain**(*domain=None*)

Change or query the current global domain. If *domain* is `None`, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

gettext.**gettext**(*message*)

Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

gettext.**dgettext**(*domain*, *message*)

Like *gettext()*, but look the message up in the specified *domain*.

gettext.**ngettext**(*singular*, *plural*, *n*)

Like *gettext()*, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in `.po` files and the formulas for a variety of languages.

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)

Like *ngettext()*, but look the message up in the specified *domain*.

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)

Similar to the corresponding functions without the `p` in the prefix (that is, *gettext()*, *dgettext()*, *ngettext()*, *dngettext()*), but the translation is restricted to the given message *context*.

在 3.8 版被加入.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 Class-based API

The class-based API of the *gettext* module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a *GNUTranslations* class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this class can also install themselves in the built-in namespace as the function `_()`.

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what *textdomain()* takes. Optional *localedir* is as in *bindtextdomain()*. Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used.[2] If *languages* is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a

[2] 請見上方*bindtextdomain()* 之🄵解。

colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

localedir/*language*/LC_MESSAGES/*domain*.mo

The first such file name that exists is returned by *find()*. If no such file is found, then None is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

Return a *Translations instance based on the *domain*, *localedir*, and *languages*, which are first passed to *find()* to get a list of the associated .mo file paths. Instances with identical .mo file names are cached. The actual class instantiated is *class_* if provided, otherwise *GNUTranslations*. The class's constructor must take a single *file object* argument.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, *copy.copy()* is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no .mo file is found, this function raises *OSError* if *fallback* is false (which is the default), and returns a *NullTranslations* instance if *fallback* is true.

在 3.3 版的變更: *IOError* 曾經被引發，現在它是一個*OSError* 的🄵名。

在 3.11 版的變更: *codeset* 參數被移除。

gettext.**install**(*domain*, *localedir=None*, *, *names=None*)

This installs the function _() in Python's builtins namespace, based on *domain* and *localedir* which are passed to the function *translation()*.

For the *names* parameter, please see the description of the translation object's *install()* method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the _() function, like this:

```
print(_('This string will be translated.'))
```

For convenience, you want the _() function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

在 3.11 版的變更: *names* is now a keyword-only parameter.

The NullTranslations class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is *NullTranslations*; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of NullTranslations:

class gettext.**NullTranslations**(*fp=None*)

Takes an optional *file object* *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through *add_fallback()*. It then calls self._parse(fp) if *fp* is not None.

_parse(*fp*)

No-op in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(*message*)

If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return *message*. Overridden in derived classes.

ngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return *singular* if *n* is 1; return *plural* otherwise. Overridden in derived classes.

pgettext(*context*, *message*)

If a fallback has been set, forward *pgettext()* to the fallback. Otherwise, return the translated message. Overridden in derived classes.

在 3.8 版被加入.

npgettext(*context*, *singular*, *plural*, *n*)

If a fallback has been set, forward *npgettext()* to the fallback. Otherwise, return the translated message. Overridden in derived classes.

在 3.8 版被加入.

info()

Return a dictionary containing the metadata found in the message catalog file.

charset()

Return the encoding of the message catalog file.

install(*names=None*)

This method installs *gettext()* into the built-in namespace, binding it to _.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'`, `'ngettext'`, `'pgettext'`, and `'npgettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

在 3.8 版的變更: 新增 `'pgettext'` 與 `'npgettext'`。

The `GNUTranslations` class

The `gettext` module provides one additional class derived from *NullTranslations*: *GNUTranslations*. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format.

GNUTranslations parses optional metadata out of the translation catalog. It is convention with GNU **gettext** to include metadata as the translation for the empty string. This metadata is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

If the `.mo` file's magic number is invalid, the major version number is unexpected, or if other problems occur while reading the file, instantiating a *GNUTranslations* class can raise *OSError*.

class gettext.**GNUTranslations**

The following methods are overridden from the base class implementation:

gettext(*message*)

Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's *gettext()* method. Otherwise, the *message* id is returned.

ngettext(*singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *ngettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

以下是個範例:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

Look up the *context* and *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id and *context*, and a fallback has been set, the look up is forwarded to the fallback's *pgettext()* method. Otherwise, the *message* id is returned.

在 3.8 版被加入.

npgettext(*context*, *singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use.

If the message id for *context* is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *npgettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

在 3.8 版被加入.

Solaris message catalog support

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the *gettext* module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

For compatibility with this older module, the function `Catalog()` is an alias for the *translation()* function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

24.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language-specific translations of the message catalogs
4. use the *gettext* module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` --- that is, a call to the function `_`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his po-utils package.

(Python also includes pure-Python versions of these programs, called **pygettext.py** and **msgfmt.py**; some Python distributions will install them for you. **pygettext.py** is similar to **xgettext**, but only understands Python source code and cannot handle other programming languages such as C or C++. **pygettext.py** supports a command-line interface similar to **xgettext**; for details on its use, run `pygettext.py --help`. **msgfmt.py** is binary compatible with GNU **msgfmt**. With these two programs, you may not need the GNU **gettext** package to internationalize your Python applications.)

xgettext, **pygettext**, and similar tools generate `.po` files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these `.po` files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a `<language-name>.po` file that's compiled into a machine-readable `.mo` binary catalog file using the **msgfmt** program. The `.mo` files are used by the *gettext* module for the actual translation processing at run-time.

How you use the *gettext* module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU **gettext** API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory, you can pass it into the *install()* function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don't actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify "a" as being translatable to the **gettext** program, because the parameter is not a string literal.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

In this case, you are marking translatable strings with the function `N_()`, which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **xgettext**, **pygettext**, `pybabel extract`, and **xpot** all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

24.1.4 致謝

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

F解

24.2 locale --- 國際化服務

原始碼： Lib/locale.py

The *locale* module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The *locale* module is implemented on top of the _locale module, which in turn uses an ANSI C locale implementation if available.

The *locale* module defines the following exception and functions:

exception locale.**Error**

Exception raised when the locale passed to *setlocale()* is not recognized.

locale.**setlocale**(*category*, *locale=None*)

If *locale* is given and not None, *setlocale()* modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a *string*, or a pair, language code and encoding. An empty string specifies the user's default settings. If the modification of the locale fails, the exception *Error* is raised. If successful, the new locale setting is returned.

If *locale* is a pair, it is converted to a locale name using the locale aliasing engine. The language code has the same format as a *locale name*, but without encoding and @-modifier. The language code and encoding can be None.

If *locale* is omitted or None, the current setting for *category* is returned.

setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

locale.**localeconv**()

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

分類	Key	含義
LC_NUMERIC	'decimal_point'	Decimal point character.
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with *CHAR_MAX*, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	Character used between groups.
LC_MONETARY	'int_curr_symbol'	International currency symbol.
	'currency_symbol'	Local currency symbol.
	'p_cs_precedes/n_cs_precedes'	Whether the currency symbol precedes the value (for positive resp. negative values).
	'p_sep_by_space/n_sep_by_space'	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	'mon_decimal_point'	Decimal point used for monetary values.
	'frac_digits'	Number of fractional digits used in local formatting of monetary values.
	'int_frac_digits'	Number of fractional digits used in international formatting of monetary values.
	'mon_thousands_sep'	Group separator used for monetary values.
	'mon_grouping'	Equivalent to 'grouping', used for monetary values.
	'positive_sign'	Symbol used to annotate a positive monetary value.
	'negative_sign'	Symbol used to annotate a negative monetary value.
	'p_sign_posn/n_sign_posn'	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to *CHAR_MAX* to indicate that there is no value specified in this locale.

The possible values for 'p_sign_posn' and 'n_sign_posn' are given below.

Value	Explanation
0	Currency and value are surrounded by parentheses.
1	The sign should precede the value and currency symbol.
2	The sign should follow the value and currency symbol.
3	The sign should immediately precede the value.
4	The sign should immediately follow the value.
CHAR_MAX	Nothing is specified in this locale.

The function temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

在 3.7 版的變更: The function now temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale in some cases.

locale.**nl_langinfo**(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The *nl_langinfo()* function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.**CODESET**

Get a string with the name of the character encoding used in the selected locale.

locale.**D_T_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent date and time in a locale-specific way.

locale.**D_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent a date in a locale-specific way.

locale.**T_FMT**

Get a string that can be used as a format string for *time.strftime()* to represent a time in a locale-specific way.

locale.**T_FMT_AMPM**

Get a format string for *time.strftime()* to represent time in the am/pm format.

locale.**DAY_1**

locale.**DAY_2**

locale.**DAY_3**

locale.**DAY_4**

locale.**DAY_5**

locale.**DAY_6**

locale.**DAY_7**

Get the name of the n-th day of the week.

> **備🄵**
>
> This follows the US convention of *DAY_1* being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

locale.**ABDAY_1**

locale.**ABDAY_2**

locale.**ABDAY_3**

locale.**ABDAY_4**

locale.**ABDAY_5**

locale.**ABDAY_6**

locale.**ABDAY_7**

Get the abbreviated name of the n-th day of the week.

locale.**MON_1**

locale.**MON_2**

locale.**MON_3**

locale.**MON_4**

locale.**MON_5**

locale.**MON_6**

locale.**MON_7**
locale.**MON_8**
locale.**MON_9**
locale.**MON_10**
locale.**MON_11**
locale.**MON_12**
Get the name of the n-th month.

locale.**ABMON_1**
locale.**ABMON_2**
locale.**ABMON_3**
locale.**ABMON_4**
locale.**ABMON_5**
locale.**ABMON_6**
locale.**ABMON_7**
locale.**ABMON_8**
locale.**ABMON_9**
locale.**ABMON_10**
locale.**ABMON_11**
locale.**ABMON_12**
Get the abbreviated name of the n-th month.

locale.**RADIXCHAR**
Get the radix character (decimal dot, decimal comma, etc.).

locale.**THOUSEP**
Get the separator character for thousands (groups of three digits).

locale.**YESEXPR**
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

locale.**NOEXPR**
Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

> **備🄵**
>
> The regular expressions for *YESEXPR* and *NOEXPR* use syntax suitable for the `regex` function from the C library, which might differ from the syntax used in *re*.

locale.**CRNCYSTR**
Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.**ERA**
Get a string which describes how years are counted and displayed for each era in a locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the *time.strftime()* function to use this information. The format of the returned string is specified in *The Open Group Base Specifications Issue 8*, paragraph 7.3.5.2 LC_TIME C-Language Access.

locale.**ERA_D_T_FMT**

Get a format string for *time.strftime()* to represent date and time in a locale-specific era-based way.

locale.**ERA_D_FMT**

Get a format string for *time.strftime()* to represent a date in a locale-specific era-based way.

locale.**ERA_T_FMT**

Get a format string for *time.strftime()* to represent a time in a locale-specific era-based way.

locale.**ALT_DIGITS**

Get a string consisting of up to 100 semicolon-separated symbols used to represent the values 0 to 99 in a locale-specific way. In most locales this is an empty string.

The function temporarily sets the `LC_CTYPE` locale to the locale of the category that determines the requested value (`LC_TIME`, `LC_NUMERIC`, `LC_MONETARY` or `LC_MESSAGES`) if locales are different and the resulting string is non-ASCII. This temporary change affects other threads.

在 3.14 版的變更: The function now temporarily sets the `LC_CTYPE` locale in some cases.

locale.**getdefaultlocale**([*envvars*])

Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

The language code has the same format as a *locale name*, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

Deprecated since version 3.11, will be removed in version 3.15.

locale.**getlocale**(*category=LC_CTYPE*)

Returns the current setting for the given locale category as a tuple containing the language code and encoding. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

The language code has the same format as a *locale name*, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

locale.**getpreferredencoding**(*do_setlocale=True*)

Return the *locale encoding* used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

On Android or if the *Python UTF-8 Mode* is enabled, always return `'utf-8'`, the *locale encoding* and the *do_setlocale* argument are ignored.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

在 3.7 版的變更: The function now always returns `"utf-8"` on Android or if the *Python UTF-8 Mode* is enabled.

locale.**getencoding**()

Get the current *locale encoding*:

- On Android and VxWorks, return `"utf-8"`.
- On Unix, return the encoding of the current *LC_CTYPE* locale. Return `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string: for example, if the current LC_CTYPE locale is not supported.
- On Windows, return the ANSI code page.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

This function is similar to *getpreferredencoding(False)* except this function ignores the *Python UTF-8 Mode*.

在 3.11 版被加入.

locale.**normalize**(*localename*)

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like *setlocale()*.

locale.**strcoll**(*string1*, *string2*)

Compares two strings according to the current *LC_COLLATE* setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

locale.**strxfrm**(*string*)

Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.**format_string**(*format*, *val*, *grouping=False*, *monetary=False*)

Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the `%` operator. For floating-point values, the decimal point is modified if appropriate. If *grouping* is `True`, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

在 3.7 版的變更: The *monetary* keyword parameter was added.

locale.**currency**(*val*, *symbol=True*, *grouping=False*, *international=False*)

Formats a number *val* according to the current *LC_MONETARY* settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

> **備🄵**
>
> This function will not work with the 'C' locale, so you have to set a locale via *setlocale()* first.

locale.**str**(*float*)

Formats a floating-point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale.**delocalize**(*string*)

Converts a string into a normalized number string, following the *LC_NUMERIC* settings.

在 3.5 版被加入.

locale.**localize**(*string*, *grouping=False*, *monetary=False*)

Converts a normalized number string into a formatted string following the *LC_NUMERIC* settings.

在 3.10 版被加入.

locale.**atof**(*string*, *func=float*)

Converts a string to a number, following the *LC_NUMERIC* settings, by calling *func* on the result of calling *delocalize()* on *string*.

locale.**atoi**(*string*)

Converts a string to an integer, following the *LC_NUMERIC* conventions.

locale.**LC_CTYPE**

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See **PEP 538** and **PEP 540** for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn't internally use locale-dependent character transformation functions from `ctype.h`. Instead, an internal `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

locale.**LC_COLLATE**

Locale category for sorting strings. The functions *strcoll()* and *strxfrm()* of the *locale* module are affected.

locale.**LC_TIME**

Locale category for the formatting of time. The function *time.strftime()* follows these conventions.

locale.**LC_MONETARY**

Locale category for formatting of monetary values. The available options are available from the *localeconv()* function.

locale.**LC_MESSAGES**

Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by *os.strerror()* might be affected by this category.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

locale.**LC_NUMERIC**

Locale category for formatting numbers. The functions *format_string()*, *atoi()*, *atof()* and *str()* of the *locale* module are affected by that category. All other numeric formatting operations are not affected.

locale.**LC_ALL**

Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.**CHAR_MAX**

This is a symbolic constant used for different values returned by *localeconv()*.

範例:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
```

(繼續下一頁)

(繼續上一頁)

```
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

24.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementations are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user's preferred locale is. There is one exception: the *LC_CTYPE* category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call *setlocale()* in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with *time.strftime()*), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: *atof()*, *atoi()*, *format_string()*, *str()*.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

24.2.2 Locale names

The format of the locale name is platform dependent, and the set of supported locales can depend on the system configuration.

On Posix platforms, it usually has the format[1]:

```
language ["_" territory] ["." charset] ["@" modifier]
```

where *language* is a two- or three-letter language code from ISO 639, *territory* is a two-letter country or region code from ISO 3166, *charset* is a locale encoding, and *modifier* is a script name, a language subtag, a sort order identifier, or other locale modifier (for example, "latin", "valencia", "stroke" and "euro").

On Windows, several formats are supported.[2][3] A subset of IETF BCP 47 tags:

```
language ["-" script] ["-" territory] ["." charset]
language ["-" script] "-" territory "-" modifier
```

where *language* and *territory* have the same meaning as in Posix, *script* is a four-letter script code from ISO 15924, and *modifier* is a language subtag, a sort order identifier or custom modifier (for example, "valencia", "stroke" or "x-python"). Both hyphen (`'-'`) and underscore (`'_'`) separators are supported. Only UTF-8 encoding is allowed for BCP 47 tags.

Windows also supports locale names in the format:

[1] IEEE Std 1003.1-2024; 8.2 Internationalization Variables
[2] UCRT Locale names, Languages, and Country/Region strings
[3] Locale Names

```
language ["_" territory] ["." charset]
```

where *language* and *territory* are full names, such as ”English” and ”United States”, and *charset* is either a code page number (for example, ”1252”) or UTF-8. Only the underscore separator is supported in this format.

The ”C” locale is supported on all platforms.

24.2.3 For extension writers and programs that embed Python

Extension modules should never call *setlocale()*, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is `C`).

When Python code uses the *locale* module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.

24.2.4 Access to message catalogs

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

`locale.`**`bind_textdomain_codeset`**(*domain*, *codeset*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions *gettext()*, *dgettext()*, *dcgettext()*, *textdomain()*, *bindtextdomain()*, and *bind_textdomain_codeset()*. These are similar to the same functions in the *gettext* module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use *gettext* instead. A known exception to this rule are applications that link with additional C libraries which internally invoke C functions `gettext` or `dcgettext`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER 25

以 Tk 打造圖形使用者介面 (Graphical User Interfaces)

Tk/Tcl 長期以來一直是 Python 不可或缺的一部分。它提供了一個🄵大且獨立於平台的視窗工具包，可供使用*tkinter* 套件及其擴充套件*tkinter.ttk* 模組的 Python 開發者使用。

tkinter 套件是 Tcl/Tk 之上的一個輕薄物件導向層。要使用*tkinter*，你不需要編寫 Tcl 程式，但會需要查🄵 Tk 文件和部份 Tcl 文件。*tkinter* 是一組將 Tk 小工具 (widget) 實作🄵 Python 類🄵的包裝器。

tkinter 的主要優點是速度快，而且通常與 Python 捆綁 (bundle) 在一起。🄵管其標准文件不是很完整，但還是有些不錯的材料，包括：參考資料、教學、書籍等。*tkinter* 曾因其過時的外觀而🄵所皆知，但這在 Tk 8.5 中得到了極大的改進。此外，還有許多其他你可能會感興趣的 GUI 函式庫。Python wiki 列出了幾個替代的 GUI 框架和工具。

25.1 `tkinter` --- Tcl/Tk 的 Python 介面

原始碼： Lib/tkinter/__init__.py

tkinter 套件（「Tk 介面」）是 Tcl/Tk GUI 工具集的標准 Python 介面。Tk 和*tkinter* 在大多數 Unix 平台（包含 macOS）以及 Windows 系統上都可以使用。

從命令列執行 `python -m tkinter` 應該會開🄵一個展示簡單 Tk 介面的視窗，讓你知道*tkinter* 已正確安裝在你的系統上，🄵且也會顯示已安裝的 Tcl/Tk 版本，這樣你就可以🄵讀該版本專屬的 Tcl/Tk 文件。

Tkinter 支援一系列的 Tcl/Tk 版本，這些版本可以是有或🄵有執行緒支援的建構版本。官方的 Python 二進位發行版捆綁了有執行緒支援的 Tcl/Tk 8.6。關於支援版本的更多資訊，請參🄵*_tkinter* 模組的原始碼。

Tkinter 🄵非一個輕薄的包裝器，而是加入了相當多自身的邏輯，使其體驗更具 Python 風格（pythonic）。本文件將著重於這些新增和變更，對於未變更的細節，則會引導讀者參🄵官方的 Tcl/Tk 文件。

備🄵

Tcl/Tk 8.5（2007）引入了一套現代化的主題式使用者介面元件，以及一個用以操作它們的新 API。新舊 API 目前都還能使用。你在網路上找到的大多數文件仍使用舊版 API，而且可能已經嚴重過時。

也參考

- **TkDocs**
  關於使用 Tkinter 建立使用者介面的詳盡教學。□容解釋了關鍵概念，□使用現代 API 來□明建議的方法。
- **Tkinter 8.5 reference: a GUI for Python**
  Tkinter 8.5 的參考文件，詳細□明了可用的類□、方法和選項。

Tcl/Tk 相關資源：

- **Tk 指令**
  Tkinter 所使用的每個底層 Tcl/Tk 指令的綜合參考資料。
- **Tcl/Tk 首頁**
  額外的文件，以及連至 Tcl/Tk 核心開發的連結。

書籍：

- **Modern Tkinter for Busy Python Developers**
  由 Mark Roseman 所著。（ISBN 978-1999149567）
- **Python GUI programming with Tkinter**
  由 Alan D. Moore 所著。（ISBN 978-1788835886）
- **Programming Python**
  由 Mark Lutz 所著；對 Tkinter 有極佳的涵蓋。（ISBN 978-0596158101）
- **Tcl and the Tk Toolkit (2nd edition)**
  由 Tcl/Tk 發明者 John Ousterhout 與 Ken Jones 所著；不包含 Tkinter。（ISBN 978-0321336330）

25.1.1 架構

Tcl/Tk □非單一函式庫，而是由幾個不同的模組所組成，每個模組都有獨立的功能和官方文件。Python 的二進位發行版也隨附了一個附加模組。

Tcl
Tcl 是一個動態直譯的程式語言，就像 Python 一樣。雖然它可以作□通用程式語言獨立使用，但它最常被嵌入到 C 應用程式中，作□□本引擎或 Tk 工具集的介面。Tcl 函式庫有一個 C 介面，可用於建立和管理一個或多個 Tcl 直譯器的實例，在這些實例中執行 Tcl 指令和□本，□新增以 Tcl 或 C 實作的自訂指令。每個直譯器都有一個事件□列，□有設施可以向其傳送事件□加以處理。與 Python 不同，Tcl 的執行模型是圍繞著協同多工（cooperative multitasking）設計的，而 Tkinter 則彌合了這種差□（詳情請參□*Threading model*）。

Tk
Tk 是一個以 C 實作的 Tcl 套件，它新增了自訂指令來建立和操作 GUI 元件。每個 `Tk` 物件都嵌入了自己載入 Tk 的 Tcl 直譯器實例。Tk 的元件非常可自訂，但代價是外觀較□過時。Tk 使用 Tcl 的事件□列來□生和處理 GUI 事件。

Ttk
主題式 Tk（Ttk）是較新的一族 Tk 元件，相較於許多傳統的 Tk 元件，它在不同平台上提供了更好的外觀。從 Tk 8.5 版開始，Ttk 作□ Tk 的一部分發行。Python 的□結（bindings）則在一個獨立的模組 `tkinter.ttk` 中提供。

在□部，Tk 和 Ttk 使用底層作業系統的設施，即 Unix/X11 上的 Xlib、macOS 上的 Cocoa、Windows 上的 GDI。

當你的 Python 應用程式使用 Tkinter 中的類□（例如，建立一個元件）時，`tkinter` 模組會先組合一個 Tcl/Tk 指令字串。它將該 Tcl 指令字串傳遞給一個□部的 `_tkinter` 二進位模組，後者再呼叫 Tcl 直譯器來對其進行計算。Tcl 直譯器接著會呼叫 Tk 和/或 Ttk 套件，而這些套件又會再呼叫 Xlib、Cocoa 或 GDI。

25.1.2 Tkinter 模組

對 Tkinter 的支援分散在數個模組中。大多數應用程式將需要主要的*tkinter*模組，以及*tkinter.ttk*模組，後者提供了現代主題式元件集和 API：

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

建構一個頂層 Tk 元件，它通常是應用程式的主視窗，🄵🄵此元件初始化一個 Tcl 直譯器。每個實例都有其自己關聯的 Tcl 直譯器。

*Tk*類🄵通常使用所有預設值來實例化。然而，目前可辨識下列關鍵字引數：

screenName
當給定（作🄵字串）時，設定 DISPLAY 環境變數。（僅限 X11）

baseName
設定檔的名稱。預設情🄵下，*baseName* 是從程式名稱（`sys.argv[0]`）衍生而來。

className
元件類🄵的名稱。用作設定檔，也用作呼叫 Tcl 時的名稱（*interp* 中的 *argv0*）。

useTk
若🄵 True，則初始化 Tk 子系統。*tkinter.Tcl()* 函式會將此設定🄵 False。

sync
若🄵 True，則同步執行所有 X 伺服器指令，以便立即回報錯誤。可用於偵錯。（僅限 X11）

use
指定要嵌入應用程式的視窗 *id*，而不是將其建立🄵獨立的頂層視窗。*id* 必須以與頂層元件的 -use 選項值相同的方式指定（也就是🄵，其形式類似 `winfo_id()` 回傳的形式）。

請注意，在某些平台上，只有當 *id* 指向一個已🄵用 -container 選項的 Tk 框架或頂層視窗時，此功能才能正常運作。

Tk 會讀取名🄵 *.className*.tcl 和 *.baseName*.tcl 的設定檔，🄵在 Tcl 直譯器中進行直譯，🄵對 *.className*.py 和 *.baseName*.py 的🄵容呼叫*exec()*。設定檔的路徑是 HOME 環境變數，如果未定義，則是*os.curdir*。

tk
透過實例化*Tk* 建立的 Tk 應用程式物件。這提供了對 Tcl 直譯器的存取。每個附加到同一個*Tk*實例的元件，其*tk* 屬性都具有相同的值。

master
包含此元件的元件物件。對於*Tk*，*master* 是*None*，因🄵它是主視窗。*master* 和 *parent* 這兩個術語很相似，有時會作🄵引數名稱互🄵使用；然而，呼叫 `winfo_parent()` 會回傳元件名稱的字串，而*master* 則回傳物件。*parent*/*child* 反映了樹狀關🄵，而 *master*/*slave* 反映了容器結構。

children
此元件的直接子代，形式🄵一個*dict*，其中鍵🄵子元件的名稱，值🄵子實例物件。

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

Tcl() 函式是一個工廠函式，它建立的物件與*Tk* 類🄵建立的物件非常相似，只是它不會初始化 Tk 子系統。這在驅動 Tcl 直譯器時最🄵有用，尤其是在不希望建立額外頂層視窗的環境中，或者在無法建立的環境中（例如🄵有 X 伺服器的 Unix/Linux 系統）。由*Tcl()* 物件建立的物件可以透過呼叫其 `loadtk()` 方法來建立一個頂層視窗（🄵初始化 Tk 子系統）。

提供 Tk 支援的模組包括：

tkinter
主要的 Tkinter 模組。

tkinter.colorchooser
讓使用者選擇🄵色的對話方塊。

tkinter.commondialog
此處列出的其他模組中定義的對話方塊的基底類🄵。

tkinter.filedialog
讓使用者指定要開🄵或儲存的檔案的通用對話方塊。

tkinter.font
協助處理字型的工具程式。

tkinter.messagebox
存取標准的 Tk 對話方塊。

tkinter.scrolledtext
🄵建垂直🄵軸的文字元件。

tkinter.simpledialog
基本的對話方塊和便利函式。

tkinter.ttk
在 Tk 8.5 中引入的主題式元件集，🄵主*tkinter* 模組中的許多傳統元件提供了現代化的替代方案。

額外模組：

_tkinter
一個包含 Tcl/Tk 低階介面的二進位模組。它由主*tkinter* 模組自動引入，應用程式開發者不應直接使用它。它通常是一個共享函式庫（或 DLL），但在某些情🄵下可能會與 Python 直譯器🄵態連結。

idlelib
Python 的整合開發與學習環境（IDLE）。基於*tkinter*。

tkinter.constants
在將各種參數傳遞給 Tkinter 呼叫時，可用於替代字串的符號常數。由主*tkinter* 模組自動引入。

tkinter.dnd
（實驗性）對*tkinter* 的拖放支援。當它被 Tk DND 取代時，將會被🄵用。

turtle
在 Tk 視窗中的海龜繪圖。

25.1.3 Tkinter 應急指南

本節🄵非旨在成🄵 Tk 或 Tkinter 的詳盡教學。🄵此，請參🄵前面提到的外部資源之一。相反地，本節提供了一個快速導覽，介紹 Tkinter 應用程式的外觀、識🄵基礎的 Tk 概念，🄵解釋 Tkinter 包裝器的結構。

本節的其余部分將幫助你識🄵在 Tkinter 應用程式中所需的類🄵、方法和選項，以及在哪🄵可以找到關於它們的更詳細文件，包括在官方的 Tcl/Tk 參考手🄵中。

一個 Hello World 程式

我們將從一個 Tkinter 的"Hello World" 應用程式開始。這不是我們能寫出的最小程式，但足以🄵明一些你需要知道的關鍵概念。

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

在引入之後，下一行建立了一個 `Tk` 類🄵的實例，它會初始化 Tk 🄵建立其關聯的 Tcl 直譯器。它還會建立一個頂層視窗，稱🄵根視窗，作🄵應用程式的主視窗。

接下來的一行建立了一個框架元件，在本例中，它將包含我們接下來要建立的一個標🄵和一個按鈕。該框架被放置在根視窗🄵。

下一行建立了一個包含🄵態文字字串的標🄵元件。`grid()` 方法用於指定標🄵在其包含的框架元件中的相對🄵局（位置），類似於 HTML 中表格的運作方式。

然後建立一個按鈕元件，🄵放置在標🄵的右側。當按下時，它將呼叫根視窗的 `destroy()` 方法。

最後，`mainloop()` 方法將所有🄵容顯示在螢幕上，🄵回應使用者輸入，直到程式終止。

重要的 Tk 概念

即使是這個簡單的程式也🄵明了以下關鍵的 Tk 概念：

元件（widgets）
: 一個 Tkinter 使用者介面是由個🄵的元件所組成。每個元件都表示🄵一個 Python 物件，從 `ttk.Frame`、`ttk.Label` 和 `ttk.Button` 等類🄵實例化而來。

元件階層（widget hierarchy）
: 元件被安排在一個階層中。標🄵和按鈕被包含在一個框架🄵，而框架又被包含在根視窗🄵。在建立每個子元件時，其父元件會作🄵第一個引數傳遞給元件的建構函式。

設定選項（configuration options）
: 元件具有設定選項，可以修改其外觀和行🄵，例如在標🄵或按鈕中顯示的文字。不同類🄵的元件會有不同的選項集。

🄵局管理（geometry management）
: 元件在建立時不會自動新增到使用者介面中。像 `grid` 這樣的🄵局管理器（geometry manager）會控制它們在使用者介面中的放置位置。

事件🄵圈（event loop）
: Tkinter 只有在主動執行事件🄵圈時，才會對使用者輸入、程式的變更，甚至螢幕刷新做出反應。如果你的程式🄵有執行事件🄵圈，你的使用者介面將不會更新。

了解 Tkinter 如何包裝 Tcl/Tk

當你的應用程式使用 Tkinter 的類🄵和方法時，Tkinter 🄵部正在組合代表 Tcl/Tk 指令的字串，🄵在附加到你應用程式 `Tk` 實例的 Tcl 直譯器中執行這些指令。

無論是試圖🄵覽參考文件、尋找正確的方法或選項、改寫一些現有程式碼，還是偵錯你的 Tkinter 應用程式，在某些時候，了解那些底層 Tcl/Tk 指令的樣貌會很有幫助。

🄵了🄵明，以下是上面 Tkinter 🄵本主要部分的 Tcl/Tk 對等程式碼。

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

Tcl 的語法類似於許多 shell 語言，第一個字是要執行的指令，後面跟著該指令的引數，以空格分隔。在不深入太多細節的情🄴下，請注意以下幾點：

- 用於建立元件的指令（如 `ttk::frame`）對應於 Tkinter 中的元件類🄵。
- Tcl 元件選項（如 `-text`）對應於 Tkinter 中的關鍵字引數。
- 在 Tcl 中，元件是透過路徑名稱（如 `.frm.btn`）來參照，而 Tkinter 不使用名稱，而是使用物件參照。
- 元件在元件階層中的位置被編碼在其（階層式）路徑名稱中，該路徑名稱使用 `.`（點）作🄵路徑分隔符。根視窗的路徑名稱就是 `.`（點）。在 Tkinter 中，階層不是由路徑名稱定義的，而是在建立每個子元件時指定父元件來定義的。
- 在 Tcl 中實作🄵獨立指令的操作（如 `grid` 或 `destroy`），在 Tkinter 中則表示🄵元件物件上的方法。正如你稍後會看到的，在其他時候，Tcl 使用的看起來像是對元件物件的方法呼叫，這更接近 Tkinter 中的用法。

我該如何... ? 哪個選項可以... ?

如果你不確定如何在 Tkinter 中做某件事，𠀀且無法立即在你正在使用的教學或參考文件中找到它，這𠀀有一些可能有幫助的策略。

首先，請記住，個𠀀元件的運作細節可能會因 Tkinter 和 Tcl/Tk 的不同版本而𠀀。如果你正在搜尋文件，請確保它對應於你系統上安裝的 Python 和 Tcl/Tk 版本。

在搜尋如何使用 API 時，知道你正在使用的類𠀀、選項或方法的確切名稱會很有幫助。自省（Introspection），無論是在互動式 Python shell 中還是使用*print()*，都可以幫助你識𠀀所需𠀀容。

要找出任何元件上有哪些可用的設定選項，請呼叫其 `configure()` 方法，該方法會回傳一個字典，其中包含有關每個物件的各種資訊，包括其預設值和目前值。使用 `keys()` 僅取得每個選項的名稱。

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

由於大多數元件有許多共通的設定選項，找出特定於某個元件類𠀀的選項會很有用。將選項串列與像框架這樣較簡單的元件的選項串列進行比較，是做到這一點的一種方法。

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

同樣地，你可以使用標準的*dir()*函式來尋找元件物件可用的方法。如果你試一下，你會看到有超過 200 個常見的元件方法，所以再次𠀀調，識𠀀特定於某個元件類𠀀的方法是很有幫助的。

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

𠀀覽 Tcl/Tk 參考手𠀀

如前所述，官方的 Tk 指令 參考手𠀀（man pages）通常是對元件上特定操作作用的最准確描述。即使你知道所需選項或方法的名稱，你可能仍有幾個地方需要查看。

雖然 Tkinter 中的所有操作都實作𠀀對元件物件的方法呼叫，但你已經看到許多 Tcl/Tk 操作顯示𠀀指令，這些指令將元件路徑名稱作𠀀其第一個參數，後面跟著可選的參數，例如：

```
destroy .
grid .frm.btn -column 0 -row 0
```

然而，其他的看起來更像是對元件物件呼叫的方法（事實上，當你在 Tcl/Tk 中建立一個元件時，它會建立一個以元件路徑名稱𠀀名的 Tcl 指令，該指令的第一個參數是要呼叫的方法名稱）。

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

在官方的 Tcl/Tk 參考文件中，你會在特定元件的 man page 上找到大多數看起來像方法呼叫的操作（例如，你可以在 ttk::button man page 上找到 `invoke()` 方法），而將元件作𠀀參數的函式通常有自己的 man page（例如 grid）。

你可以在 options 或 ttk::widget man page 中找到許多常見的選項和方法，而其他的則在特定元件類𠀀的 man page 中找到。

你還會發現許多 Tkinter 方法有𠀀合名稱，例如 `winfo_x()`、`winfo_height()`、`winfo_viewable()`。你可以在 winfo man page 中找到所有這些文件的𠀀明。

備𠀀

有點令人困惑的是，所有 Tkinter 元件上還有一些方法，它們實際上𠀀不是對該元件進行操作，而是在全域範圍𠀀操作，獨立於任何元件。例如存取剪貼簿或系統鈴聲的方法。（它們恰好被實作𠀀所有 Tkinter 元件都繼承的基底 `Widget` 類𠀀中的方法）。

25.1.4 執行緒模型

Python 和 Tcl/Tk 有非常不同的執行緒模型，*tkinter* 試圖彌合這種差F。如果你使用執行緒，你可能需要意識到這一點。

一個 Python 直譯器可能有多個與之關聯的執行緒。在 Tcl 中，可以建立多個執行緒，但每個執行緒都有一個獨立的 Tcl 直譯器實例與之關聯。執行緒也可以建立多個直譯器實例，但每個直譯器實例只能由建立它的那個執行緒使用。

由*tkinter* 建立的每個 `Tk` 物件都包含一個 Tcl 直譯器。它還會追F是哪個執行緒建立了該直譯器。可以從任何 Python 執行緒呼叫*tkinter*。在F部，如果呼叫來自建立 `Tk` 物件的執行緒以外的執行緒，則會將一個事件發F到直譯器的事件F列中，執行後，結果會回傳給呼叫的 Python 執行緒。

Tcl/Tk 應用程式通常是事件驅動的，這意味著在初始化之後，直譯器會執行一個事件F圈（即 `Tk.mainloop()`）F回應事件。因F它是單執行緒的，事件處理常式必須快速回應，否則它們會阻塞其他事件的處理。F避免這種情F，任何長時間執行的計算都不應在事件處理常式中執行，而是使用計時器將其分解成更小的部分，或在另一個執行緒中執行。這與許多 GUI 工具集不同，在那些工具集中，GUI 在一個與所有應用程式碼（包括事件處理常式）完全分離的執行緒中執行。

如果 Tcl 直譯器F有在執行事件F圈和處理事件，那F從執行 Tcl 直譯器的執行緒以外的執行緒發出的任何*tkinter* 呼叫都將失敗。

存在一些特殊情F：

- Tcl/Tk 函式庫可以被建構成不具備執行緒感知能力。在這種情F下，*tkinter* 會從原始的 Python 執行緒呼叫該函式庫，即使這與建立 Tcl 直譯器的執行緒不同。一個全域鎖確保一次只發生一個呼叫。
- 雖然*tkinter* 允許你建立多個 `Tk` 物件的實例（每個都有自己的直譯器），但屬於同一個執行緒的所有直譯器共享一個共同的事件F列，這很快就會變得混亂。在實務上，不要一次建立多個 `Tk` 的實例。否則，最好在不同的執行緒中建立它們，F確保你正在執行一個具備執行緒感知能力的 Tcl/Tk 建構版本。
- 阻塞事件處理常式F不是防止 Tcl 直譯器重新進入事件F圈的唯一方法。甚至可以執行多個巢狀的事件F圈或完全放F事件F圈。如果你在事件或執行緒方面做任何F雜的操作，請注意這些可能性。
- 目前有少數幾個*tkinter* 函式只有在從建立 Tcl 直譯器的執行緒中呼叫時才能正常運作。

25.1.5 實用參考

設定選項

選項控制著諸如元件的F色和邊框寬度等。選項可以透過三種方式設定：

在物件建立時，使用關鍵字引數

```
fred = Button(self, fg="red", bg="blue")
```

在物件建立後，將選項名稱視F字典索引

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

在物件建立後，使用 config() 方法更新多個屬性

```
fred.config(fg="red", bg="blue")
```

有關給定選項及其行F的完整解釋，請參F相關元件的 Tk man pages。

請注意，man pages F每個元件列出了「標準選項」和「元件特定選項」。前者是許多元件共有的選項串列，後者是該特定元件特有的選項。標準選項記F在 *options(3)* man page 中。

本文件中未對標準選項和元件特定選項進行區分。某些選項不適用於某些類型的元件。給定元件是否回應特定選項取F於該元件的類F；按鈕有 `command` 選項，標F則F有。

給定元件支援的選項列在該元件的 man page 中，或者可以在 runtime 透過不帶引數呼叫 `config()` 方法，或對該元件呼叫 `keys()` 方法來查詢。這些呼叫的回傳值是一個字典，其鍵是選項名稱的字串（例如 `'relief'`），其值是 5-tuples。

某些選項，如 `bg`，是具有長名稱的常見選項的同義詞（`bg` 是"background" 的簡寫）。將簡寫選項的名稱傳遞給 `config()` 方法將回傳一個 2-tuple，而不是 5-tuple。回傳的 2-tuple 將包含同義詞的名稱和「真實」選項（例如 `('bg', 'background')`）。

索引	含義	範例
0	選項名稱	`'relief'`
1	用於資料庫查詢的選項名稱	`'relief'`
2	用於資料庫查詢的選項類🄵	`'Relief'`
3	預設值	`'raised'`
4	目前值	`'groove'`

範例：

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

當然，印出的字典將包含所有可用的選項及其值。這僅作🄵一個範例。

Packer 🄵局管理器

packer 是 Tk 的🄵局管理（geometry manager）機制之一。🄵局管理器（geometry manager）用於指定元件在其容器（它們共同的 *master*）🄵的相對位置。與較🄵繁瑣的 *placer*（較不常用，我們在此不予介紹）相比，packer 🄵用定性的關🄵規範——上方、左方、填滿等——🄵🄵你計算出確切的放置座標。

任何 *master* 元件的大小由其🄵部的「slave 元件」的大小🄵定。packer 用於控制 slave 元件在被 packed 的 master 🄵部出現的位置。你可以將元件 pack 到框架中，再將框架 pack 到其他框架中，以達到你想要的🄵局。此外，一旦 packed，🄵局會動態調整以適應設定的增量變更。

請注意，元件在透過🄵局管理器（geometry manager）指定其🄵局（geometry）之前是不會出現的。一個常見的早期錯誤是忽略了🄵局（geometry）指定，然後在元件被建立但🄵有任何東西出現時感到驚訝。一個元件只有在例如 packer 的 `pack()` 方法被應用於其上之後才會出現。

pack() 方法可以與關鍵字選項/值對一起呼叫，以控制元件在其容器中出現的位置，以及當主應用程式視窗調整大小時它的行🄵方式。以下是一些範例：

```
fred.pack()                     # 預設🄵 side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Packer 選項

有關 packer 及其可接受選項的更詳盡資訊，請參🄵 man pages 和 John Ousterhout 書中的第 183 頁。

anchor
: 錨點類型。表示 packer 將每個 slave 放置在其區域中的位置。

expand
: 布林值，`0` 或 `1`。

fill
: 合法值：`'x'`、`'y'`、`'both'`、`'none'`。

ipadx 和 ipady
: 一個距離——指定 slave 元件每側的🄵部填充。

padx 和 pady
: 一個距離——指定 slave 元件每側的外部填充。

side
合法值🄵：`'left'`、`'right'`、`'top'`、`'bottom'`。

耦合元件變數

某些元件（如文字輸入元件）的當前值設定可以透過使用特殊選項直接連接到應用程式變數。這些選項是 `variable`、`textvariable`、`onvalue`、`offvalue` 和 `value`。這種連接是雙向的：如果變數因任何原因發生變化，與之連接的元件將會更新以反映新值。

不幸的是，在*tkinter* 的目前實作中，無法透過 `variable` 或 `textvariable` 選項將任意的 Python 變數傳遞給元件。唯一適用此功能的變數類型是從*tkinter* 中定義的一個名🄵 Variable 的類🄵繼承的子類🄵變數。

已經定義了許多有用的 Variable 子類🄵：`StringVar`、`IntVar`、`DoubleVar` 和 `BooleanVar`。要讀取此類變數的目前值，請對其呼叫 `get()` 方法，要更改其值，則呼叫 `set()` 方法。如果你遵循此協定，元件將始終追🄵變數的值，無需你進一步干預。

舉例來🄵：

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # 建立應用程式變數。
        self.contents = tk.StringVar()
        # 將其設定🄵某個值。
        self.contents.set("this is a variable")
        # 告訴 entry 元件監視此變數。
        self.entrythingy["textvariable"] = self.contents

        # 🄵使用者按下 return 鍵定義一個回呼。
        # 它會印出變數的目前值。
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

視窗管理器

在 Tk 中，有一個工具指令 `wm`，用於與視窗管理器互動。`wm` 指令的選項允許你控制標題、位置、圖示點陣圖等。在*tkinter* 中，這些指令已實作🄵 `Wm` 類🄵上的方法。頂層元件是 `Wm` 類🄵的子類🄵，因此可以直接呼叫 `Wm` 方法。

要取得包含給定元件的頂層視窗，你通常可以直接參照該元件的 master。當然，如果該元件被 packed 在一個框架🄵，master 將不代表一個頂層視窗。要取得包含任意元件的頂層視窗，你可以呼叫 `_root()` 方法。此方法以底🄵開頭，表示此函式是實作的一部分，而不是 Tk 功能的介面。

以下是一些常見用法範例：

```
import tkinter as tk
```

(繼續下一頁)

(繼續上一頁)

```
class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# 建立應用程式
myapp = App()

#
# 以下是對視窗管理器類🄵的方法呼叫
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# 🄵動程式
myapp.mainloop()
```

Tk 選項資料型🄵

anchor
合法值🄵羅盤方位點：`"n"`、`"ne"`、`"e"`、`"se"`、`"s"`、`"sw"`、`"w"`、`"nw"`，以及 `"center"`。

bitmap
有八個🄵建的、命名的點陣圖：`'error'`、`'gray25'`、`'gray50'`、`'hourglass'`、`'info'`、`'questhead'`、`'question'`、`'warning'`。要指定一個 X 點陣圖檔名，請給出檔案的完整路徑，🄵在前面加上一個 `@`，例如 `"@/usr/contrib/bitmap/gumby.bit"`。

boolean
你可以傳遞整數 0 或 1，或字串 `"yes"` 或 `"no"`。

callback
這是任何不帶引數的 Python 函式。例如：

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color
🄵色可以以 rgb.txt 檔案中的 X 🄵色名稱給出，或以表示 RGB 值的字串給出，範圍可以是 4 位元：`"#RGB"`、8 位元：`"#RRGGBB"`、12 位元：`"#RRRGGGBBB"` 或 16 位元：`"#RRRRGGGGBBBB"`，其中 R、G、B 在此代表任何合法的十六進位數字。詳情請參🄵 Ousterhout 書中的第 160 頁。

cursor
可以使用 `cursorfont.h` 中的標準 X 游標名稱，無需 `XC_` 前綴。例如，要取得手形游標（`XC_hand2`），請使用字串 `"hand2"`。你也可以指定自己的點陣圖和遮罩檔案。請參🄵 Ousterhout 書中的第 179 頁。

distance
螢幕距離可以以像素或🄵對距離指定。像素以數字給出，🄵對距離以字串給出，尾隨字元表示單位：`c` 代表公分，`i` 代表英寸，`m` 代表毫米，`p` 代表印刷的點。例如，3.5 英寸表示🄵 `"3.5i"`。

font
Tk 使用串列字型名稱格式，例如 `{courier 10 bold}`。帶有正數的字型大小以點🄵單位；帶有負數的大小以像素🄵單位。

geometry
這是一個 `widthxheight` 形式的字串，其中 width 和 height 對於大多數元件是以像素🄵單位（對於顯示文字的元件則以字元🄵單位）。例如：`fred["geometry"] = "200x100"`。

justify
合法值🄵字串：`"left"`、`"center"`、`"right"` 和 `"fill"`。

region
這是一個由四個以空格分隔的元素組成的字串，每個元素都是一個合法的距離（見上文）。例如：

`"2 3 4 5"`、`"3i 2i 4.5i 2i"` 和 `"3c 2c 4c 10.43c"` 都是合法的區域。

relief
🄵定元件的邊框樣式。合法值🄵：`"raised"`、`"sunken"`、`"flat"`、`"groove"` 和 `"ridge"`。

scrollcommand
這幾乎總是某個🄵軸元件的 `set()` 方法，但也可以是任何接受單一引數的元件方法。

wrap
必須是以下之一：`"none"`、`"char"` 或 `"word"`。

🄵結與事件

來自元件指令的 bind 方法允許你監視某些事件，🄵在該事件類型發生時觸發一個回呼函式。bind 方法的形式是：

```
def bind(self, sequence, func, add=''):
```

其中：

sequence
是一個表示目標事件類型的字串。（詳情請參🄵 *bind(3tk)* man page，以及 John Ousterhout 的書 *Tcl and the Tk Toolkit (2nd edition)* 的第 201 頁）。

func
是一個 Python 函式，接受一個引數，在事件發生時被呼叫。一個 Event 實例將作🄵引數傳遞。（以這種方式部署的函式通常稱🄵回呼。）

add
是可選的，可以是 `''` 或 `'+'`。傳遞一個空字串表示此🄵結將取代與此事件關聯的任何其他🄵結。傳遞一個 `'+'` 意味著此函式將被新增到🄵結到此事件類型的函式串列中。

舉例來🄵：

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

請注意在 `turn_red()` 回呼中如何存取事件的 widget 欄位。此欄位包含捕獲 X 事件的元件。下表列出了你可以存取的其他事件欄位，以及它們在 Tk 中的表示方式，這在參🄵 Tk man pages 時可能很有用。

Tk	Tkinter 事件欄位	Tk	Tkinter 事件欄位
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index 參數

許多元件需要傳遞"index" 參數。這些參數用於指向 Text 元件中的特定位置、Entry 元件中的特定字元，或 Menu 元件中的特定選單項目。

Entry 元件索引（index、view index 等）
Entry 元件具有參照所顯示文字中字元位置的選項。你可以使用這些 *tkinter* 函式來存取文字元件中的這些特殊點：

Text 元件索引
Text 元件的索引表示法非常豐富，在 Tk man pages 中有最好的描述。

選單索引（menu.invoke()、menu.entryconfig() 等）

選單的某些選項和方法會操作特定的選單項目。每當選項或參數需要選單索引時，你可以傳入：

- 一個整數，它參照元件中項目的數字位置，從頂部開始計數，從 0 開始；
- 字串 `"active"`，它參照目前在游標下的選單位置；
- 字串 `"last"`，它參照最後一個選單項目；
- 一個以 `@` 🄵前綴的整數，如 `@6`，其中該整數被解釋🄵選單座標系統中的 y 像素座標；
- 字串 `"none"`，表示完全🄵有選單項目，最常用於 menu.activate() 以停用所有項目，最後，
- 一個文字字串，它會與從選單頂部掃描到底部的選單項目標🄴進行模式匹配。請注意，此索引類型在所有其他類型之後考慮，這意味著標🄵🄵 `last`、`active` 或 `none` 的選單項目的匹配可能會被解釋🄵上述的字面值。

圖片

可以透過 `tkinter.Image` 的相應子類🄵來建立不同格式的圖片：

- `BitmapImage` 用於 XBM 格式的圖片。
- `PhotoImage` 用於 PGM、PPM、GIF 和 PNG 格式的圖片。後者從 Tk 8.6 開始支援。

任一類型的圖片都是透過 `file` 或 `data` 選項建立的（也有其他選項可用）。

在 3.13 版的變更: 新增了 `PhotoImage` 方法 `copy_replace()`，用於將一個區域從一個圖片🄵🄵到另一個圖片，可能帶有像素縮放和/或子🄵樣。將 *from_coords* 參數新增到 `PhotoImage` 方法 `copy()`、`zoom()` 和 `subsample()`。將 *zoom* 和 *subsample* 參數新增到 `PhotoImage` 方法 `copy()`。

然後，圖片物件可以在任何支援 `image` 選項的元件（例如標🄴、按鈕、選單）中使用。在這些情🄵下，Tk 將不會保留對圖片的參照。當對圖片物件的最後一個 Python 參照被🄵除時，圖片資料也會被🄴除，Tk 將在使用該圖片的任何地方顯示一個空框。

也參考

Pillow 套件新增了對 BMP、JPEG、TIFF 和 WebP 等格式的支援。

25.1.6 檔案處理常式

Tk 允許你🄵🄵和取消🄵🄵一個回呼函式，當檔案描述器上可以進行 I/O 時，該函式將從 Tk 主🄵圈中被呼叫。每個檔案描述器只能🄵🄵一個處理常式。範例程式碼：

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

此功能在 Windows 上不可用。

由於你不知道有多少位元組可供讀取，你可能不想使用 *BufferedIOBase* 或 *TextIOBase* 的 *read()* 或 *readline()* 方法，因🄴這些方法會堅持讀取預定義數量的位元組。對於 sockets（通訊端），*recv()* 或 *recvfrom()* 方法可以正常運作；對於其他檔案，請使用原始讀取或 `os.read(file.fileno(), maxbytecount)`。

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

🄵🄵檔案處理常式回呼函式 *func*。*file* 引數可以是一個具有 *fileno()* 方法的物件（例如檔案或 socket（通訊端）物件），或是一個整數檔案描述器。*mask* 引數是下面三個常數中任意一個的 OR 組合。回呼函式被呼叫的方式如下：

```
callback(file, mask)
```

Widget.tk.**deletefilehandler**(*file*)

取消🄵🄵檔案處理常式。

_tkinter.**READABLE**

_tkinter.**WRITABLE**

_tkinter.**EXCEPTION**

用於 *mask* 引數的常數。

25.2 tkinter.colorchooser --- 🄵色選擇對話框

原始碼： Lib/tkinter/colorchooser.py

---

tkinter.colorchooser 模組提供類🄵*Chooser* 當作與原生🄵色選擇器對話框的介面。Chooser 實作了一個🄵色選擇的互動視窗。類🄵 Chooser 繼承了類🄵*Dialog*。

class tkinter.colorchooser.**Chooser**(*master=None*, ***options*)

tkinter.colorchooser.**askcolor**(*color=None*, ***options*)

建立一個🄵色選擇對話框。一旦呼叫這個方法便會顯示視窗，等待使用者做出選擇後，回傳選擇的🄵色（或者是 None）給呼叫者。

也參考

***tkinter.commondialog* 模組**
Tkinter 標准對話框模組

25.3 tkinter.font --- Tkinter 字型包裝器

原始碼： Lib/tkinter/font.py

---

tkinter.font 模組提供類🄵*Font*，可以建立及使用已命名的字型。

不同的字重 (font weights) 以及傾斜 (slant) 是：

tkinter.font.**NORMAL**

tkinter.font.**BOLD**

tkinter.font.**ITALIC**

tkinter.font.**ROMAN**

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

類🄵*Font* 代表一個已命名字型。*Font* 實例會被賦予一個的名字，也可以特指他們的字型家族 (font family)、字級 (size)、以及外觀設定。已命名字型是 Tk 建立及辨識字型🄵單一物件的方式，而不是每次出現時特指字型的屬性。

引數：

font - 字型指定符號元組 (family, size, options)
name - 獨特字型名稱
exists - 如果存在的話，指向現有的已命名字型

額外的關鍵字選項（若已指定 *font* 則會忽略）：

family - 字型家族，例如：Courier、Times
size - 字級
如果 *size* 是正數則會直譯成以點 (point) 𝔽單位的字級。
如果 *size* 是負數則會變成𝔽對值
以像素 (pixel) 𝔽單位的字級。
weight - 𝔽調字型，例如：NORMAL（標准體）、BOLD（粗體）
slant - 例如：ROMAN（正體）、ITALIC（斜體）
underline - 字型加上底𝔽（0 - 無底𝔽、1 - 加上底𝔽）
overstrike - 字型加上𝔽除𝔽（0 - 無𝔽除𝔽、1 - 加上𝔽除𝔽）

actual(*option=None*, *displayof=None*)
回傳字型的屬性。

cget(*option*)
取得字型的其中一個屬性。

config(***options*)
修改字體的多個屬性。

copy()
回傳目前字體的新實例。

measure(*text*, *displayof=None*)
回傳目前字型被格式化時，在特定顯示區域中文字所用的空間。若顯示區域𝔽有被指定，則會假定主程式視窗𝔽顯示區域。

metrics(**options*, ***kw*)
回傳字型特定的資料。其選項包含：

***ascent* - 基准𝔽以及最高點的距離**
在字型中的一個字母可以𝔽用的空間

***descent* - 基准𝔽以及最低點的距離**
在字型中的一個字母可以𝔽用的空間

***linespace* - 最小所需的垂直間距**
在字型中的任兩個字母之間，確保跨行時不會有垂直重𝔽。

fixed - 若字型𝔽等寬 (fixed-width) 的則𝔽 1，否則𝔽 0

tkinter.font.**families**(*root=None*, *displayof=None*)
回傳不同的字型家族。

tkinter.font.**names**(*root=None*)
回傳已定義字型的名字。

tkinter.font.**nametofont**(*name*, *root=None*)
回傳一個*Font*，代表一個 tk 已命名字型。

在 3.10 版的變更: 新增 *root* 參數。

25.4 Tkinter 對話框

25.4.1 tkinter.simpledialog --- 標準 Tkinter 輸入對話框

原始碼： Lib/tkinter/simpledialog.py

The *tkinter.simpledialog* module contains convenience classes and functions for creating simple modal dialogs to get a value from the user.

tkinter.simpledialog.**askfloat**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askinteger**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askstring**(*title*, *prompt*, ***kw*)

The above three functions provide dialogs that prompt the user to enter a value of the desired type.

class tkinter.simpledialog.**Dialog**(*parent*, *title=None*)

The base class for custom dialogs.

body(*master*)

Override to construct the dialog's interface and return the widget that should have initial focus.

buttonbox()

Default behaviour adds OK and Cancel buttons. Override for custom button layouts.

25.4.2 tkinter.filedialog --- File selection dialogs

原始碼： Lib/tkinter/filedialog.py

The *tkinter.filedialog* module provides classes and factory functions for creating file/directory selection windows.

Native Load/Save Dialogs

The following classes and functions provide file dialog windows that combine a native look-and-feel with configuration options to customize behaviour. The following keyword arguments are applicable to the classes and functions listed below:

parent - the window to place the dialog on top of

title - the title of the window

initialdir - the directory that the dialog starts in

initialfile - the file selected upon opening of the dialog

filetypes - a sequence of (label, pattern) tuples, '*' wildcard is allowed

defaultextension - default extension to append to file (save dialogs)

multiple - when true, selection of multiple items is allowed

Static factory functions

The below functions when called create a modal, native look-and-feel dialog, wait for the user's selection, then return the selected value(s) or `None` to the caller.

tkinter.filedialog.**askopenfile**(*mode='r'*, ***options*)

tkinter.filedialog.**askopenfiles**(*mode='r'*, ***options*)

The above two functions create an *Open* dialog and return the opened file object(s) in read-only mode.

tkinter.filedialog.**asksaveasfile**(*mode='w'*, ***options*)

Create a *SaveAs* dialog and return a file object opened in write-only mode.

tkinter.filedialog.**askopenfilename**(***options*)

tkinter.filedialog.**askopenfilenames**(***options*)

The above two functions create an *Open* dialog and return the selected filename(s) that correspond to existing file(s).

tkinter.filedialog.**asksaveasfilename**(***options*)

Create a *SaveAs* dialog and return the selected filename.

tkinter.filedialog.**askdirectory**(***options*)

Prompt user to select a directory.

Additional keyword option:

mustexist - determines if selection must be an existing directory.

class tkinter.filedialog.**Open**(*master=None*, ***options*)

class tkinter.filedialog.**SaveAs**(*master=None*, ***options*)

The above two classes provide native dialog windows for saving and loading files.

Convenience classes

The below classes are used for creating file/directory windows from scratch. These do not emulate the native look-and-feel of the platform.

class tkinter.filedialog.**Directory**(*master=None*, ***options*)

Create a dialog prompting the user to select a directory.

備🄵

The *FileDialog* class should be subclassed for custom event handling and behaviour.

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)

Create a basic file selection dialog.

cancel_command(*event=None*)

Trigger the termination of the dialog window.

dirs_double_event(*event*)

Event handler for double-click event on directory.

dirs_select_event(*event*)

Event handler for click event on directory.

files_double_event(*event*)

Event handler for double-click event on file.

files_select_event(*event*)

Event handler for single-click event on file.

filter_command(*event=None*)

Filter the files by directory.

get_filter()

Retrieve the file filter currently in use.

get_selection()

Retrieve the currently selected item.

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)

Render dialog and start event loop.

ok_event(*event*)

Exit dialog returning current selection.

quit(*how=None*)

Exit dialog returning filename, if any.

set_filter(*dir*, *pat*)

Set the file filter.

set_selection(*file*)

Update the current file selection to *file*.

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting an existing file.

ok_command()

Test that a file is provided and that the selection indicates an already existing file.

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting a destination file.

ok_command()

Test whether or not the selection points to a valid file that is not a directory. Confirmation is required if an already existing file is selected.

25.4.3 tkinter.commondialog --- Dialog window templates

原始碼： Lib/tkinter/commondialog.py

The *tkinter.commondialog* module provides the *Dialog* class that is the base class for dialogs defined in other supporting modules.

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(***options*)

Render the Dialog window.

也參考

tkinter.messagebox 模組、tut-files

25.5 tkinter.messagebox --- Tkinter 訊息提示

原始碼： Lib/tkinter/messagebox.py

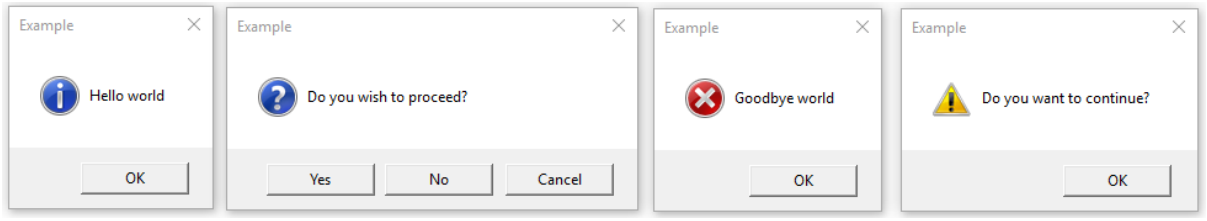
---

tkinter.messagebox 模組提供了模板基底類🄴以及各種常用配置的便捷方法。訊息框 (message box) 是互動視窗 (modal)，會基於使用者的選擇回傳（`True`、`False`、`None`、*OK*、*CANCEL*、*YES*、*NO*）的子集。常見的訊息框樣式 (style) 和版面配置 (layout) 包括但不限於：

class tkinter.messagebox.**Message**(*master=None*, ***options*)

建立一個訊息視窗，其中包含應用程式指定的訊息、一個圖示和一組按鈕。訊息視窗中的每個按鈕都有唯一的符號名稱作🄴識🄴（請參考 *type* 選項）。

支援以下選項：



command
指定當使用者關閉對話框 (dialog) 時要呼叫的函式。使用者按一下以關閉對話框的按鈕的名稱作🄵引數傳遞。此選項僅適用於 macOS。

default
給出此訊息視窗的預設按鈕的符號名稱（*OK*、*CANCEL* 等）。如果未指定此選項，則對話框中的第一個按鈕將成🄵預設按鈕。

detail
透過 *message* 選項指定將輔助訊息給主訊息。訊息詳細資訊將顯示在主要訊息下方，🄵且在作業系統支援的情🄵下，將以比主要訊息更不🄵調的字體顯示。

icon
指定要顯示的圖示。如果未指定此選項，則會顯示 *INFO* 圖示。

message
指定要在此訊息框中顯示的訊息。預設值🄵空字串。

parent
使指定視窗成🄵訊息框的邏輯父視窗 (logical parent window)。訊息框顯示在其父視窗的頂部。

title
指定顯示🄵訊息框標題的字串。此選項在 macOS 上被忽略，其平台指南禁止在此類對話方塊上使用標題。

type
安排一組需顯示的預先定義的按鈕組合。

show(***options*)
顯示訊息視窗🄵等待使用者選擇其中一個按鈕。然後回傳所選按鈕的符號名稱。關鍵字引數可以覆寫建構函式中指定的選項。

資訊訊息框

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)
建立🄵顯示具有指定標題和訊息的資訊訊息框。

警告訊息框

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)
建立🄵顯示具有指定標題和訊息的警告訊息框。

tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)
建立🄵顯示具有指定標題和訊息的錯誤訊息框。

問題留言框

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***, *type=YESNO*, ***options*)
問一個問題。預設顯示按鈕 *YES* 和 *NO*。回傳所選按鈕的符號名稱。

tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)
詢問操作是否應該繼續。顯示按鈕 *OK* 和 *CANCEL*。如果答案正確則傳回 `True`，否則回傳 `False`。

tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)
詢問是否應重試操作。顯示按鈕 *RETRY* 和 *CANCEL*。如果答案🄵是，則回傳 `True`，否則回傳 `False`。

tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)

問一個問題。顯示按鈕*YES*和*NO*。如果答案🄵是，則回傳 `True`，否則回傳 `False`。

tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)

問一個問題。顯示按鈕*YES*、*NO*和*CANCEL*。如果答案🄵是，則回傳 `True`；如果取消則回傳 `None`，否則回傳 `False`。

按鈕的符號名稱：

tkinter.messagebox.**ABORT = 'abort'**

tkinter.messagebox.**RETRY = 'retry'**

tkinter.messagebox.**IGNORE = 'ignore'**

tkinter.messagebox.**OK = 'ok'**

tkinter.messagebox.**CANCEL = 'cancel'**

tkinter.messagebox.**YES = 'yes'**

tkinter.messagebox.**NO = 'no'**

預先定義的按鈕組合：

tkinter.messagebox.**ABORTRETRYIGNORE = 'abortretryignore'**

顯示三個按鈕，其符號名稱🄵*ABORT*、*RETRY*和*IGNORE*。

tkinter.messagebox.**OK = 'ok'**

顯示一個按鈕，其符號名稱🄵*OK*。

tkinter.messagebox.**OKCANCEL = 'okcancel'**

顯示兩個按鈕，其符號名稱🄵*OK*和*CANCEL*。

tkinter.messagebox.**RETRYCANCEL = 'retrycancel'**

顯示兩個按鈕，其符號名稱🄵*RETRY*和*CANCEL*。

tkinter.messagebox.**YESNO = 'yesno'**

顯示兩個按鈕，其符號名稱🄵*YES*和*NO*。

tkinter.messagebox.**YESNOCANCEL = 'yesnocancel'**

顯示三個按鈕，其符號名稱🄵*YES*、*NO*和*CANCEL*。

圖示圖像：

tkinter.messagebox.**ERROR = 'error'**

tkinter.messagebox.**INFO = 'info'**

tkinter.messagebox.**QUESTION = 'question'**

tkinter.messagebox.**WARNING = 'warning'**

25.6 `tkinter.scrolledtext` --- 🄵動文字小工具

原始碼： Lib/tkinter/scrolledtext.py

The *tkinter.scrolledtext* module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the ”right thing.” Using the *ScrolledText* class is a lot easier than setting up a text widget and scroll bar directly.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the *`ScrolledText`* widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

class `tkinter.scrolledtext.`**ScrolledText**(*master=None*, ***kw*)

frame

The frame which surrounds the text and scroll bar widgets.

vbar

The scroll bar widget.

25.7 `tkinter.dnd` --- 拖放支援

原始碼： Lib/tkinter/dnd.py

備🄵

這是一個實驗性的功能，🄵將在被 Tk DND 取代時🄵用。

`tkinter.dnd` 模組🄵單一應用程式🄵、同一個視窗🄵或視窗之間的物件提供拖放支援。要讓一個物件可以被拖動，你必須🄵它建立一個事件🄵結來🄵動拖放過程。你通常會將一個 ButtonPress 事件🄵結到你所寫的回呼函式（請參🄵🄵*結與事件*）。該函式應該呼叫*`dnd_start()`*，其中'source' 是要被拖動的物件，而'event' 是觸發呼叫的事件（你的回呼函式的引數）。

目標物件的選擇過程如下：

1. 由上而下搜尋滑鼠下方的區域以尋找目標元件（widget）
   - 目標元件應該要有一個可呼叫的 *dnd_accept* 屬性（attribute）
   - 如果 *dnd_accept* 不存在或回傳 `None`，搜尋會移至父元件
   - 如果找不到目標元件，則目標物件🄵 `None`
2. 呼叫 *<old_target>.dnd_leave(source, event)*
3. 呼叫 *<new_target>.dnd_enter(source, event)*
4. 呼叫 *<target>.dnd_commit(source, event)* 以通知放下動作
5. 呼叫 *<source>.dnd_end(target, event)* 以表示拖放結束

class `tkinter.dnd.`**DndHandler**(*source*, *event*)

DndHandler 類🄵處理拖放事件，它會追🄵事件元件的根上的 Motion 和 ButtonRelease 事件。

cancel(*event=None*)

取消拖放過程。

finish(*event*, *commit=0*)

執行拖放結束函式。

on_motion(*event*)

在拖動過程中檢查滑鼠下方的區域以尋找目標物件。

on_release(*event*)

當釋放模式被觸發時，發出拖動結束的訊號。

tkinter.dnd.**dnd_start**(*source*, *event*)

用於拖放過程的工廠函式。

也參考

🄵結與事件

25.8 tkinter.ttk --- Tk 主題化小工具

原始碼： Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. It provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

也參考

Tk Widget Styling Support
A document introducing theming support for Tk

25.8.1 使用 Ttk

To start using Ttk, import its module:

```
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

也參考

Converting existing applications to use Tile widgets
A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

25.8.2 Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar`, and *Spinbox*. The other six are new: *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` and *Treeview*. And all them are subclasses of *Widget*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling*, see the `Style` class documentation.

25.8.3 Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

標準選項

All the `ttk` Widgets accept the following options:

選項	描述
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

選項	描述
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

選項	描述
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by *Style.map()*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

相容性選項

選項	描述
state	May be set to ”normal” or ”disabled” to control the ”disabled” state bit. This is a write-only option: setting it changes the widget state, but the *Widget.state()* method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

旗標	描述
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	”On”, ”true”, or ”current” for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an ”active” or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget’s value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class tkinter.ttk.**Widget**

identify(*x*, *y*)

Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)

Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)

Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently enabled state flags.

statespec will usually be a list or a tuple.

25.8.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

選項

This widget accepts the following specific options:

選項	描述
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of ”left”, ”center”, or ”right”.
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
state	One of ”normal”, ”readonly”, or ”disabled”. In the ”readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the ”normal” state, the text field is directly editable. In the ”disabled” state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widgets generates a **«ComboboxSelected»** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)

If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()

Returns the current value of the combobox.

set(*value*)

Sets the value of the combobox to *value*.

25.8.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

選項

This widget accepts the following specific options:

選項	描述
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **«Increment»** virtual event when the user presses <Up>, and a **«Decrement»** virtual event when the user presses <Down>.

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()

Returns the current value of the spinbox.

set(*value*)

Sets the value of the spinbox to *value*.

25.8.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently displayed window.

選項

This widget accepts the following specific options:

選項	描述
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

選項	描述
state	Either ”normal”, ”disabled” or ”hidden”. If ”disabled”, then the tab is not selectable. If ”hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters ”n”, ”s”, ”e” or ”w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form ”@x,y”, which identifies the tab
- The literal string ”current”, which identifies the currently selected tab
- The literal string ”end”, which returns the number of tabs (only valid for *`Notebook.index()`*)

🄵擬事件

This widget generates a **«NotebookTabChanged»** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)

Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

可用的選項清單請見*Tab Options*。

forget(*tab_id*)

Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)

Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)

Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)

Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string ”end”.

insert(*pos*, *child*, ***kw*)

Inserts a pane at the specified position.

pos is either the string ”end”, an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

可用的選項清單請見*Tab Options*。

select(*tab_id=None*)

Selects the specified *tab_id*.

The associated child window will be displayed, and the previously selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)

Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()

Returns a list of windows managed by the notebook.

enable_traversal()

Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.
- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

25.8.7 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

選項

This widget accepts the following specific options:

選項	描述
orient	One of ”horizontal” or ”vertical”. Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	”determinate” 與”indeterminate” 其中之一
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In ”determinate” mode, this represents the amount of work completed. In ”indeterminate” mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one ”cycle” when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)

Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)

Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()

Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

25.8.8 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

選項

This widget accepts the following specific option:

選項	描述
orient	One of ”horizontal” or ”vertical”. Specifies the orientation of the separator.

25.8.9 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On macOS, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ....), the `Sizegrip` widget will not resize the window.
- This widget supports only "southeast" resizing.

25.8.10 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

選項

This widget accepts the following specific options:

選項	描述
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string ”#all”.
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of ”extended”, ”browse” or ”none”. If set to ”extended” (the default), multiple items may be selected. If ”browse”, only a single item will be selected at a time. If ”none”, the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is ”tree headings”, i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show=”tree” is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

選項	描述
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	`True`/`False` value indicating whether the item’s children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

選項	描述
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item’s image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

🄵解:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

🄵擬事件

The Treeview widget generates the following virtual events.

事件	描述
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The *Treeview.focus()* and *Treeview.selection()* methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)

Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)

Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)

Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

有效的選項/值🄵:

id
回傳欄位名稱。這是唯讀選項。

***anchor*: One of the standard Tk anchor values.**
Specifies how the text in this column should be aligned with respect to the cell.

***minwidth*: width**
The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.

***stretch*: `True/False`**
Specifies whether the column's width should be adjusted when the widget is resized.

***width*: width**
The width of the column in pixels.

To configure the tree column, call this with column = "#0"

`delete`(**items*)

Delete all specified *items* and all their descendants.

The root item may not be deleted.

`detach`(**items*)

Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

`exists`(*item*)

Returns `True` if the specified *item* is present in the tree.

`focus`(*item=None*)

If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

`heading`(*column*, *option=None*, ***kw*)

Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

有效的選項/值🄵:

***text*: text**
The text to display in the column heading.

***image*: imageName**
Specifies an image to display to the right of the column heading.

***anchor*: anchor**
Specifies how the heading text should be aligned. One of the standard Tk anchor values.

***command*: callback**
A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

`identify`(*component*, *x*, *y*)

Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

`identify_row`(*y*)

Returns the item ID of the item at position *y*.

`identify_column`(*x*)

Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

`identify_region`(*x*, *y*)

Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element(*x*, *y*)

Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index(*item*)

Returns the integer index of *item* within its parent's list of children.

insert(*parent*, *index*, *iid=None*, ***kw*)

Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

可用的選項清單請見*Item Options*。

item(*item*, *option=None*, ***kw*)

Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move(*item*, *parent*, *index*)

Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next(*item*)

Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent(*item*)

Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev(*item*)

Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach(*item*, *parent*, *index*)

一個`Treeview.move()`的🄵名。

see(*item*)

確保 *item* 是可見的。

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection()

Returns a tuple of selected items.

在 3.8 版的變更: `selection()` no longer takes arguments. For changing the selection state use the following selection methods.

selection_set(**items*)

items becomes the new selection.

在 3.6 版的變更: *items* can be passed as separate arguments, not just as a single tuple.

selection_add(**items*)

將 *items* 加入選擇。

在 3.6 版的變更: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove(**items*)

從選擇中移除 *items*。

在 3.6 版的變更: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle(**items*)

Toggle the selection state of each item in *items*.

在 3.6 版的變更: *items* can be passed as separate arguments, not just as a single tuple.

set(*item*, *column=None*, *value=None*)

With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind(*tagname*, *sequence=None*, *callback=None*)

Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure(*tagname*, *option=None*, ***kw*)

Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has(*tagname*, *item=None*)

If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(**args*)

Query or modify horizontal position of the treeview.

yview(**args*)

Query or modify vertical position of the treeview.

25.8.11 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

也參考

Tcl'2004 conference presentation
This document explains how the theme engine works

class tkinter.ttk.**Style**

This class is used to manipulate the style database.

configure(*style*, *query_opt=None*, ***kw*)

Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup(*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either ”image”, ”from” or ”vsapi”. The latter is only available in Tk 8.6 on Windows.

If ”image” is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

border=padding
: padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

height=height
: Specifies a minimum height for the element. If less than zero, the base image’s height is used as a default.

padding=padding
: Specifies the element’s interior padding. Defaults to border’s value if not specified.

sticky=spec
: Specifies how the image is placed within the final parcel. spec contains zero or more characters ”n”, ”s”, ”w”, or ”e”.

width=width
: Specifies a minimum width for the element. If less than zero, the base image’s width is used as a default.

範例：

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active', img3),
                     border=(2, 4), sticky='we')
```

If ”from” is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

範例：

```
style = ttk.Style(root)
style.element_create('plain.background', 'from', 'default')
```

If ”vsapi” is used as the value of *etype*, `element_create()` will create a new element in the current theme whose visual appearance is drawn using the Microsoft Visual Styles API which is responsible for the themed styles on Windows XP and Vista. *args* is expected to contain the Visual Styles class and part as given in the Microsoft documentation followed by an optional sequence of tuples of ttk states and the corresponding Visual Styles API state value. *kw* may have the following options:

padding=padding
: Specify the element's interior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left. In other words, a list of three numbers specify the left, vertical, and right padding; a list of two numbers specify the horizontal and the vertical padding; a single number specifies the same padding all the way around the widget. This option may not be mixed with any other options.

margins=padding
: Specifies the elements exterior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. This option may not be mixed with any other options.

width=width
: Specifies the width for the element. If this option is set then the Visual Styles API will not be queried for the recommended size or the part. If this option is set then *height* should also be set. The *width* and *height* options cannot be mixed with the *padding* or *margins* options.

height=height
: Specifies the height of the element. See the comments for *width*.

範例：

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

在 3.13 版的變更: Added support of the ”vsapi” element factory.

element_names()
: Returns the list of elements defined in the current theme.

element_options(*elementname*)
: Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
: Create a new theme.

  It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)

Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods *Style.configure()*, *Style.map()*, *Style.layout()* and *Style.element_create()* respectively.

As an example, let's change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

回傳所有已知的主題。

theme_use(*themename=None*)

If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel.

有效的選項/值🄴:

***side*: whichside**

Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.

***sticky*: nswe**

Specifies where the element is placed inside its allocated parcel.

***unit*: 0 或 1**

If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *Widget.identify()* et al. It's used for things like scrollbar thumbs with grips.

***children*: [sublayout...]**

Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

25.9 IDLE --- Python 編輯器與 shell

原始碼： Lib/idlelib/

IDLE 是 Python 的整合開發與學習環境 (Integrated Development and Learning Environment)。

IDLE 具有以下功能：

- 跨平台：在 Windows、Unix 和 macOS 上的運作大致相同
- Python shell window（互動式直譯器）提供有色彩的程式碼輸入、輸出和錯誤訊息
- 多視窗文字編輯器，具有多次復原、Python 語法上色、智慧縮排、呼叫提示、自動完成等功能
- 能在任何視窗中搜尋、編輯器視窗中取代文字，以及搜尋多個檔案 (grep)
- 具有持久性斷點、能🄵逐步執行和檢視全域和區域命名空間的除錯器
- 設定、🄵覽器和其他對話框

25.9.1 目🄵

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu (Shell and Editor)

新增檔案
Create a new file editing window.

Open...
Open an existing file with an Open dialog.

Open Module...
Open an existing module (searches sys.path).

Recent Files
Open a list of recent files. Click one to open it.

Module Browser
Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As...
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window. (If your file namager is set to hide extensions, the current extension will be omitted in the file name box. If the new filename has no '.', '.py' and '.txt' will be added for Python and text files, except that on macOS Aqua,'.py' is added for all files.)

Save Copy As...
Save the current window to different file without changing the associated file. (See Save As note above about filename extensions.)

Print Window
Print the current window to the default printer.

Close Window
Close the current window (if an unsaved editor, ask to save; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

離開 IDLE
Close all windows and quit IDLE (ask to save unsaved edit windows).

Edit menu (Shell and Editor)

Undo
Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo
Redo the last undone change to the current window.

Select All（選擇全部）
Select the entire contents of the current window.

Cut（剪下）
Copy selection into the system-wide clipboard; then delete the selection.

Copy（🄵🄵）
Copy selection into the system-wide clipboard.

Paste（貼上）
Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Find...
Open a search dialog with many options

Find Again
Repeat the last search, if there is one.

Find Selection
Search for the currently selected string, if there is one.

Find in Files...
Open a file search dialog. Put results in a new output window.

Replace...
Open a search-and-replace dialog.

Go to Line
Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions
Open a scrollable list allowing selection of existing names. See *Completions* in the Editing and navigation section below.

Expand Word
Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show Call Tip
After an unclosed parenthesis for a function, open a small window with function parameter hints. See *Calltips* in the Editing and navigation section below.

Show Surrounding Parens
Highlight the surrounding parenthesis.

Format menu (Editor window only)

Format Paragraph
Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Indent Region
Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
Insert ## in front of selected lines.

Uncomment Region
Remove leading # or ## from selected lines.

Tabify Region
Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
Turn *all* tabs into the correct number of spaces.

Toggle Tabs
Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Strip Trailing Chitespace
Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run menu (Editor window only)

Run Module
Do *Check Module*. If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Run... Customized
Same as *Run Module*, but run the module with customized settings. *Command Line Arguments* extend `sys.argv` as if passed on a command line. The module can be run in the Shell without restarting.

Check Module
Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Python Shell
Open or wake up the Python Shell window.

Shell menu (Shell window only)

View Last Restart
Scroll the shell window to the last Shell restart.

Restart Shell
Restart the shell to clean the environment and reset display and exception handling.

Previous History
Cycle through earlier commands in history which match the current entry.

Next History
Cycle through later commands in history which match the current entry.

Interrupt Execution
Stop a running program.

Debug menu (Shell window only)

Go to File/Line
Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE
Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see *Setting preferences* under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only)
Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See *Code Context* in the Editing and Navigation section below.

Show/Hide Line Numbers (Editor Window only)
Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see *Setting preferences*).

Zoom/Restore Height
Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

關於 IDLE
Display version, copyright, license, credits, and more.

IDLE Help
Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs
Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo
Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the *Help sources* subsection below for more on Help menu choices.

Context menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut（剪下）
Copy selection into the system-wide clipboard; then delete the selection.

Copy（🄵🄵）
Copy selection into the system-wide clipboard.

Paste（貼上）
Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's `.idlerc` directory.

Set Breakpoint
Set a breakpoint on the current line.

Clear Breakpoint
Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line
Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze
If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

25.9.2 Editing and Navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

The IDLE insertion cursor is a thin vertical bar between character positions. When characters are entered, the insertion cursor and everything to its right moves right one character and the new character is entered in the new space.

Several non-character keys move the cursor and possibly delete characters. Deletion does not puts text on the clipboard, but IDLE has an undo list. Wherever this doc discusses keys, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS. (And all such discussions assume that the keys have not been re-bound to something else.)

- Arrow keys move the cursor one character or line.

- C-LeftArrow and C-RightArrow moves left or right one word.
- Home and End go to the beginning or end of the line.
- Page Up and Page Down go up or down one screen.
- C-Home and C-End go to beginning or end of the file.
- Backspace and Del (or C-d) delete the previous or next character.
- C-Backspace and C-Del delete one word left or right.
- C-k deletes ('kills') everything to the right.

Standard keybindings (like C-c to copy and C-v to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the *Format menu*.

Search and Replace

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If [x] Regular expression is checked, the target is interpreted according to the Python re module.

Completions

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters; by hitting Up, Down, PageUp, PageDown, Home, and End keys; and by a single click within the box. Close the box with Escape, Enter, and double Tab keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type '.'. For filenames in the root directory, type *os.sep* or *os.altsep* immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is C-space. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting Tab after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking 'Show Completions', or hitting Tab after a prefix, outside of a string and without a preceding '.' opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes initially exclude names beginning with '_' or, for modules, not included in '__all__'. The hidden names can be accessed by typing '_' after '.', either before or after the box is opened.

Calltips

A calltip is shown automatically when one types `(` after the name of an *accessible* function. A function name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. Whenever the cursor is in the argument part of a definition, select Edit and "Show Call Tip" on the menu or enter its shortcut to display a calltip.

The calltip consists of the function's signature and docstring up to the latter's first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A '/' or '*' in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not itself import turtle. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write(` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Shell window

In IDLE's Shell, enter, edit, and recall complete statements. (Most consoles and terminals only work with a single physical line at a time).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (`\`), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`, as specified above. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a *SyntaxError* when multiple statements are compiled as if they were one.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following:

- `C-c` attempts to interrupt statement execution (but may fail).
- `C-d` closes Shell if typed at a `>>>` prompt.
- `Alt-p` and `Alt-n` (`C-p` and `C-n` on macOS) retrieve to the current prompt the previous or next previously entered statement that matches anything already typed.
- `Return` while the cursor is on any previous statement appends the latter to anything already typed at the prompt.

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

IDLE also highlights the soft keywords `match`, `case`, and _ in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some _-s in `case` patterns.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

25.9.3 Startup and Code Execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

命令列用法

IDLE can be invoked from the command line with various options. The general syntax is:

```
python -m idlelib [options] [file ...]
```

The following options are available:

-c <command>
: Run the specified Python command in the shell window. For example, pass `-c "print('Hello, World!')"`. On Windows, the outer quotes must be double quotes as shown.

-d
: Enable the debugger and open the shell window.

-e
: Open an editor window.

-h
: Print a help message with legal combinations of options and exit.

-i
: Open a shell window.

-r <file>
: Run the specified file in the shell window.

-s
: Run the startup file (as defined by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`) before opening the shell window.

-t <title>
: Set the title of the shell window.

-
: Read and execute standard input in the shell window. This option must be the last one before any arguments.

If arguments are provided:

- If `-`, `-c`, or `-r` is used, all arguments are placed in `sys.argv[1:]`, and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'` respectively. No editor window is opened, even if that is the default set in the *Options* dialog.
- Otherwise, arguments are treated as files to be opened for editing, and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it usually displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system's network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:`. The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

On Unix-based systems with tcl/tk older than `8.6.11` (see `About IDLE`) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print

faster in IDLE, format and join together everything one wants displayed together and then print a single string. Both format strings and *str.join()* can help combine fields and lines.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. (On Windows, use `python` or `py` rather than `pythonw` or `pyw`.) The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

在 3.4 版之後被🄴用.

25.9.4 Help and Preferences

Help sources

Help menu entry ”IDLE Help” displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry ”Python Docs” opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where ’x.y’ is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

設定偏好

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular

characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

macOS 上的 IDLE

Under System Preferences: Dock, one can set "Prefer tabs when opening documents" to "Always". This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

25.9.5 idlelib --- implementation of IDLE application

原始碼： Lib/idlelib/

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under 'Startup', the idlelib code is 'private' in sense that feature changes can be backported (see **PEP 434**).

25.10 `turtle` --- 龜圖學 (Turtle graphics)

原始碼： Lib/turtle.py

25.10.1 介紹

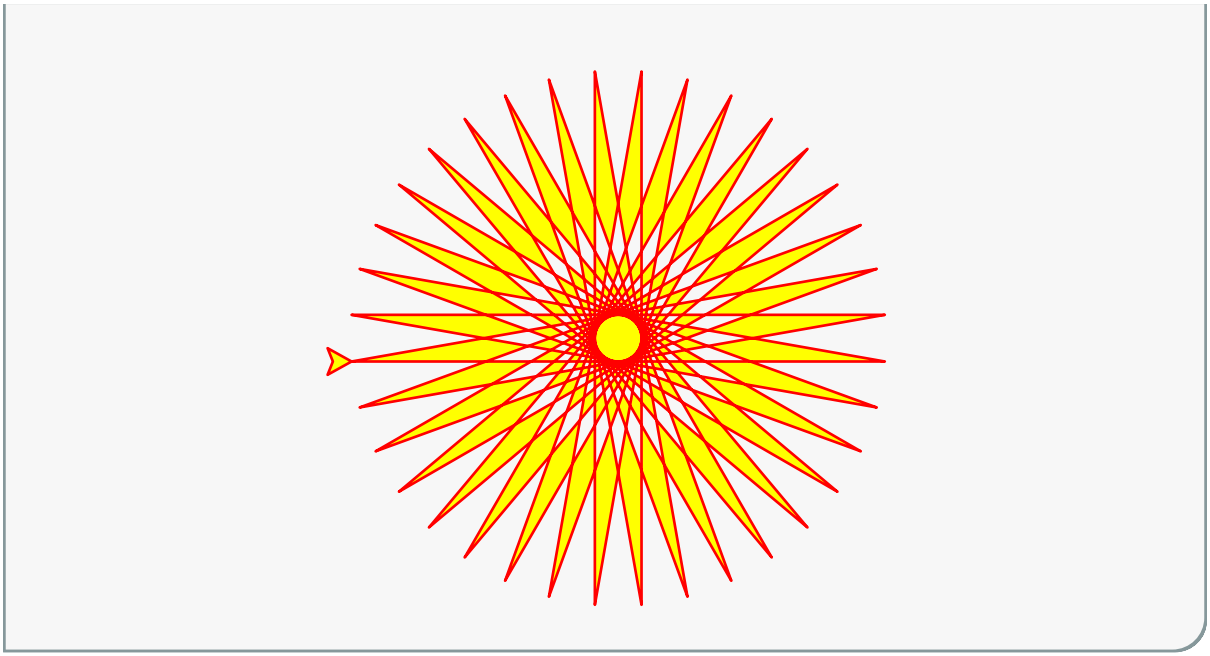
龜圖學是由 Wally Feurzeig，Seymour Papert 與 Cynthia Solomon 於 1967 年開發的，一個 以 Logo 程式語言撰寫的廣受歡迎的幾何繪圖工具。

25.10.2 Get started

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle`, give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

Turtle star

龜可以使用重🄵簡單動作之程式來畫出🄵雜的形狀。



In Python, turtle graphics provides a representation of a physical ”turtle” (a little robot with a pen) that draws on a sheet of paper on the floor.

It's an effective and well-proven way for learners to encounter programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.

Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

25.10.3 教學

New users should start here. In this tutorial we'll explore some of the basics of turtle drawing.

🄵動一個烏龜環境

在 Python shell 中，引入 `turtle` 模組中所有物件：

```
from turtle import *
```

If you run into a `No module named '_tkinter'` error, you'll have to install the *Tk interface package* on your system.

基本繪圖

Send the turtle forward 100 steps:

```
forward(100)
```

You should see (most likely, in a new window on your display) a line drawn by the turtle, heading East. Change the direction of the turtle, so that it turns 120 degrees left (anti-clockwise):

```
left(120)
```

Let's continue by drawing a triangle:

```
forward(100)
left(120)
forward(100)
```

Notice how the turtle, represented by an arrow, points in different directions as you steer it.

Experiment with those commands, and also with `backward()` and `right()`.

Pen control

Try changing the color - for example, `color('blue')` - and width of the line - for example, `width(3)` - and then drawing again.

You can also move the turtle around without drawing, by lifting up the pen: `up()` before moving. To start drawing again, use `down()`.

The turtle's position

Send your turtle back to its starting-point (useful if it has disappeared off-screen):

```
home()
```

The home position is at the center of the turtle's screen. If you ever need to know them, get the turtle's x-y coordinates with:

```
pos()
```

Home is at `(0, 0)`.

And after a while, it will probably help to clear the window so we can start anew:

```
clearscreen()
```

Making algorithmic patterns

Using loops, it's possible to build up geometric patterns:

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- which of course, are limited only by the imagination!

Let's draw the star shape at the top of this page. We want red lines, filled in with yellow:

```
color('red')
fillcolor('yellow')
```

Just as `up()` and `down()` determine whether lines will be drawn, filling can be turned on and off:

```
begin_fill()
```

Next we'll create a loop:

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` is a good way to know when the turtle is back at its home position.

Finally, complete the filling:

```
end_fill()
```

(Note that filling only actually takes place when you give the `end_fill()` command.)

25.10.4 How to...

This section covers some typical turtle use-cases and approaches.

Get started as quickly as possible

One of the joys of turtle graphics is the immediate, visual feedback that's available from simple commands - it's an excellent way to introduce children to programming ideas, with a minimum of overhead (not just children, of course).

The turtle module makes this possible by exposing all its basic functionality as functions, available with `from turtle import *`. The *turtle graphics tutorial* covers this approach.

It's worth noting that many of the turtle commands also have even more terse equivalents, such as `fd()` for *`forward()`*. These are especially useful when working with learners for whom typing is not a skill.

> You'll need to have the *`Tk interface package`* installed on your system for turtle graphics to work. Be warned that this is not always straightforward, so check this in advance if you're planning to use turtle graphics with a learner.

Automatically begin and end filling

Starting with Python 3.14, you can use the *`fill()` context manager* instead of *`begin_fill()`* and *`end_fill()`* to automatically begin and end fill. Here is an example:

```
with fill():
    for i in range(4):
        forward(100)
        right(90)

forward(200)
```

The code above is equivalent to:

```
begin_fill()
for i in range(4):
    forward(100)
    right(90)
end_fill()

forward(200)
```

Use the `turtle` module namespace

Using `from turtle import *` is convenient - but be warned that it imports a rather large collection of objects, and if you're doing anything but turtle graphics you run the risk of a name conflict (this becomes even more an issue if you're using turtle graphics in a script where other modules might be imported).

The solution is to use `import turtle` - `fd()` becomes `turtle.fd()`, `width()` becomes `turtle.width()` and so on. (If typing "turtle" over and over again becomes tedious, use for example `import turtle as t` instead.)

Use turtle graphics in a script

It's recommended to use the `turtle` module namespace as described immediately above, for example:

```
import turtle as t
from random import random

```

(繼續下一頁)

(繼續上一頁)

```
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

Another step is also required though - as soon as the script ends, Python will also close the turtle's window. Add:

```
t.mainloop()
```

to the end of the script. The script will now wait to be dismissed and will not exit until it is terminated, for example by closing the turtle graphics window.

Use object-oriented turtle graphics

也參考

Explanation of the object-oriented interface

Other than for very basic introductory purposes, or for trying things out as quickly as possible, it's more usual and much more powerful to use the object-oriented approach to turtle graphics. For example, this allows multiple turtles on screen at once.

In this approach, the various turtle commands are methods of objects (mostly of `Turtle` objects). You *can* use the object-oriented approach in the shell, but it would be more typical in a Python script.

The example above then becomes:

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)

t.screen.mainloop()
```

Note the last line. `t.screen` is an instance of the *`Screen`* that a Turtle instance exists on; it's created automatically along with the turtle.

The turtle's screen can be customised, for example:

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

25.10.5 Turtle graphics reference

備🄵

In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

Turtle methods

Turtle motion

Move and draw

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
teleport()
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Tell Turtle's state

position() | *pos()*
towards()
xcor()
ycor()
heading()
distance()

Setting and measurement

degrees()
radians()

Pen control

Drawing state

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

Color control

color()
pencolor()
fillcolor()

Filling

filling()
fill()
begin_fill()
end_fill()

More drawing control

reset()

clear()
write()

Turtle state

Visibility

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

Appearance

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Using events

onclick()
onrelease()
ondrag()

Special Turtle methods

poly()
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

Methods of TurtleScreen/Screen

Window control

bgcolor()
bgpic()
clearscreen()
resetscreen()
screensize()
setworldcoordinates()

Animation control

no_animation()
delay()
tracer()
update()

Using screen events

listen()

onkey() | *onkeyrelease()*
onkeypress()
onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

Settings and special methods

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

Input methods

textinput()
numinput()

Methods specific to Screen

bye()
exitonclick()
save()
setup()
title()

25.10.6 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

Turtle motion

`turtle.`**`forward`**`(`*distance*`)`
`turtle.`**`fd`**`(`*distance*`)`

參數
`distance` -- a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**`(`*distance*`)`
`turtle.`**`bk`**`(`*distance*`)`
`turtle.`**`backward`**`(`*distance*`)`

參數
`distance` -- a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**right**(*angle*)

turtle.**rt**(*angle*)

參數

angle -- a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*angle*)

turtle.**lt**(*angle*)

參數

angle -- a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the *degrees()* and *radians()* functions.) Angle orientation depends on the turtle mode, see *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

參數

- **x** -- a number or a pair/vector of numbers
- **y** -- a number or `None`

If *y* is `None`, *x* must be a pair of coordinates or a *Vec2D* (e.g. as returned by *pos()*).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
```

(繼續下一頁)

(繼續上一頁)

```
>>> turtle.pos()
(0.00,0.00)
```

turtle.**teleport**(*x*, *y=None*, *, *fill_gap=False*)

參數

- **x** -- a number or None
- **y** -- a number or None
- **fill_gap** -- a boolean

Move turtle to an absolute position. Unlike goto(x, y), a line will not be drawn. The turtle's orientation does not change. If currently filling, the polygon(s) teleported from will be filled after leaving, and filling will begin again after teleporting. This can be disabled with fill_gap=True, which makes the imaginary line traveled during teleporting act as a fill barrier like in goto(x, y).

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

在 3.12 版被加入.

turtle.**setx**(*x*)

參數

x -- a number (integer or float)

Set the turtle's first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

參數

y -- a number (integer or float)

Set the turtle's second coordinate to *y*, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)

turtle.**seth**(*to_angle*)

參數

to_angle -- a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees:

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

Move turtle to the origin -- coordinates (0,0) -- and set its heading to its start-orientation (which depends on the mode, see *mode()*).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

參數

- **radius** -- a number
- **extent** -- a number (or None)
- **steps** -- an integer (or None)

Draw a circle with given *radius*. The center is *radius* units left of the turtle; *extent* -- an angle -- determines which part of the circle is drawn. If *extent* is not given, draw the entire circle. If *extent* is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if *radius* is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of *extent*.

As the circle is approximated by an inscribed regular polygon, *steps* determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

參數

- **size** -- an integer >= 1 (if given)
- **color** -- a colorstring or a numeric color tuple

Draw a circular dot with diameter *size*, using *color*. If *size* is not given, the maximum of pensize+4 and 2*pensize is used.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

參數

stampid -- an integer, must be return value of previous *stamp()* call

Delete stamp with given *stampid*.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

參數

n -- an integer (or `None`)

Delete all or first/last *n* of turtle's stamps. If *n* is `None`, delete all stamps, if $n > 0$ delete first *n* stamps, else if $n < 0$ delete last *n* stamps.

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

參數

speed -- an integer in the range 0..10 or a speedstring (see below)

Set the turtle's speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: *speed* = 0 means that *no* animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle's state

turtle.**position**()

turtle.**pos**()

Return the turtle's current location (x,y) (as a *Vec2D* vector).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

參數

- **x** -- a number or a pair/vector of numbers or a turtle instance
- **y** -- a number if *x* is a number, else None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo".

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

Return the turtle's current heading (value depends on the turtle mode, see *mode()*).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

參數

- **x** -- a number or a pair/vector of numbers or a turtle instance
- **y** -- a number if *x* is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.**degrees**(*fullcircle=360.0*)

參數

fullcircle -- a number

Set angle measurement units, i.e. set number of "degrees" for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
```

(繼續下一頁)

(繼續上一頁)

```
90.0

>>> # Change angle measurement unit to grad (also known as gon,
>>> # grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

turtle.**pendown**()

turtle.**pd**()

turtle.**down**()

Pull the pen down -- drawing when moving.

turtle.**penup**()

turtle.**pu**()

turtle.**up**()

Pull the pen up -- no drawing when moving.

turtle.**pensize**(*width=None*)

turtle.**width**(*width=None*)

參數

width -- a positive number

Set the line thickness to *width* or return it. If resizemode is set to ”auto” and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

參數

- **pen** -- a dictionary with some or all of the below listed keys
- **pendict** -- one or more keyword-arguments with the below listed keys as keywords

Return or set the pen’s attributes in a ”pen-dictionary” with the following key/value pairs:

- ”shown”: True/False
- ”pendown”: True/False

- ”pencolor”: color-string or color-tuple
- ”fillcolor”: color-string or color-tuple
- ”pensize”: positive number
- ”speed”: number in range 0..10
- ”resizemode”: ”auto” or ”user” or ”noresize”
- ”stretchfactor”: (positive number, positive number)
- ”outline”: positive number
- ”tilt”: number

This dictionary can be used as argument for a subsequent call to *pen()* to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**()

Return `True` if pen is down, `False` if it’s up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Color control

turtle.**pencolor**(**args*)

Return or set the pencolor.

Four input formats are allowed:

pencolor()
Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring)
Set pencolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

pencolor((r, g, b))
Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

pencolor(r, g, b)
Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)
Return or set the fillcolor.

Four input formats are allowed:

fillcolor()
Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring)
Set fillcolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

fillcolor((r, g, b))
Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

fillcolor(r, g, b)
Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # 🄵整數而非浮點數
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)
Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color()
Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by *pencolor()* and *fillcolor()*.

color(colorstring), color((r,g,b)), color(r,g,b)
Inputs as in *pencolor()*, set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method *colormode()*.

Filling

`turtle.`**filling**`()`
Return fillstate (`True` if filling, `False` else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

`turtle.`**fill**`()`
Fill the shape drawn in the `with turtle.fill():` block.

```
>>> turtle.color("black", "red")
>>> with turtle.fill():
...     turtle.circle(80)
```

Using `fill()` is equivalent to adding the *begin_fill()* before the fill-block and *end_fill()* after the fill-block:

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

在 3.14 版被加入.

`turtle.`**begin_fill**`()`
To be called just before drawing a shape to be filled.

`turtle.`**end_fill**`()`
Fill the shape drawn after the last call to *begin_fill()*.

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.**reset**()

Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()

Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.**write**(*arg*, *move=False*, *align='left'*, *font=('Arial', 8, 'normal')*)

參數

- **arg** -- object to be written to the TurtleScreen
- **move** -- True/False
- **align** -- one of the strings "left", "center" or right"
- **font** -- a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* ("left", "center" or "right") and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

turtle.**hideturtle**()

turtle.**ht**()

Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()

turtle.**st**()

Make the turtle visible.

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

Return `True` if the Turtle is shown, `False` if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

turtle.**shape**(*name=None*)

參數

name -- a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: "arrow", "turtle", "circle", "square", "triangle", "classic". To learn about how to deal with shapes see Screen method *register_shape()*.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

參數

rmode -- one of the strings "auto", "user", "noresize"

Set resizemode to one of the values: "auto", "user", "noresize". If *rmode* is not given, return current resizemode. Different resizemodes have the following effects:

- "auto": adapts the appearance of the turtle corresponding to the value of pensize.
- "user": adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by *shapesize()*.
- "noresize": no adaption of the turtle's appearance takes place.

`resizemode("user")` is called by *shapesize()* when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

參數

- **stretch_wid** -- positive number
- **stretch_len** -- positive number
- **outline** -- positive number

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resizemode to "user". If and only if resizemode is set to "user", the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shape's outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

參數

shear -- number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle's heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

參數

angle -- a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**tiltangle**(*angle=None*)

參數

angle -- a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

參數

- **t11** -- a number (optional)
- **t12** -- a number (optional)
- **t21** -- a number (optional)
- **t12** -- a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, t22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Using events

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** -- number of the mouse-button, defaults to 1 (left mouse button)
- **add** -- `True` or `False` -- if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is `None`, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** -- number of the mouse-button, defaults to 1 (left mouse button)

- **add** -- `True` or `False` -- if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is `None`, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

`turtle.`**`ondrag`**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** -- number of the mouse-button, defaults to 1 (left mouse button)
- **add** -- `True` or `False` -- if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is `None`, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

`turtle.`**`poly`**()

Record the vertices of a polygon drawn in the `with turtle.poly():` block. The first and last vertices will be connected.

```
>>> with turtle.poly():
...     turtle.forward(100)
...     turtle.right(60)
...     turtle.forward(100)
```

在 3.14 版被加入.

`turtle.`**`begin_poly`**()

Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

`turtle.`**`end_poly`**()

Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

`turtle.`**`get_poly`**()

Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
```

(繼續下一頁)

(繼續上一頁)

```
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()

Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()

turtle.**getpen**()

Return the Turtle object itself. Only reasonable use: as a function to return the ”anonymous turtle”:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()

Return the *TurtleScreen* object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

參數

size -- 一個整數或 None

Set or disable undobuffer. If *size* is an integer, an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the *undo()* method/function. If *size* is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class *Shape* explicitly as described below:

1. Create an empty Shape object of type ”compound”.
2. Add as many components to this object as desired, using the *addcomponent()* method.

   舉例來🄵:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen's shapelist and use it:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

備🄵

The *Shape* class is used internally by the *register_shape()* method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

25.10.7 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

`turtle.`**`bgcolor`**(**args*)

參數

`args` -- a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**(*picname=None*)

參數

`picname` -- a string, name of an image file (PNG, GIF, PGM, and PPM) or `"nopic"`, or `None`

Set background image or return name of current backgroundimage. If *picname* is a filename, set the corresponding image as background. If *picname* is `"nopic"`, delete background image, if present. If *picname* is `None`, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()

備🄵

This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

turtle.**clearscreen**()

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

turtle.**reset**()

> **備🄵**
>
> This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

turtle.**resetscreen**()

Reset all Turtles on the Screen to their initial state.

turtle.**screensize**(*canvwidth=None*, *canvheight=None*, *bg=None*)

參數

- **canvwidth** -- positive integer, new width of canvas in pixels
- **canvheight** -- positive integer, new height of canvas in pixels
- **bg** -- colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

turtle.**setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

參數

- **llx** -- a number, x-coordinate of lower left corner of canvas
- **lly** -- a number, y-coordinate of lower left corner of canvas
- **urx** -- a number, x-coordinate of upper right corner of canvas
- **ury** -- a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode ”world” if necessary. This performs a `screen.reset()`. If mode ”world” is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Animation control

turtle.**no_animation**()

Temporarily disable turtle animation. The code written inside the `no_animation` block will not be animated; once the code block is exited, the drawing will appear.

```
>>> with screen.no_animation():
...     for dist in range(2, 400, 2):
...         fd(dist)
...         rt(90)
```

在 3.14 版被加入.

turtle.**delay**(*delay=None*)

參數

delay -- positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

參數

- **n** -- nonnegative integer
- **delay** -- nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of n. Second argument sets delay value (see *delay()*).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

Perform a TurtleScreen update. To be used when tracer is turned off.

另請參🄵 RawTurtle/Turtle 的*speed()* 方法。

Using screen events

turtle.**listen**(*xdummy=None*, *ydummy=None*)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass *listen()* to the onclick method.

turtle.**onkey**(*fun*, *key*)

turtle.**onkeyrelease**(*fun*, *key*)

參數

- **fun** -- a function with no arguments or `None`

- **key** -- a string: key (e.g. ”a”) or key-symbol (e.g. ”space”)

Bind *fun* to key-release event of key. If *fun* is `None`, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.`**onkeypress**(*fun*, *key=None*)

參數

- **fun** -- a function with no arguments or `None`
- **key** -- a string: key (e.g. ”a”) or key-symbol (e.g. ”space”)

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.`**onclick**(*fun*, *btn=1*, *add=None*)

`turtle.`**onscreenclick**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** -- number of the mouse-button, defaults to 1 (left mouse button)
- **add** -- `True` or `False` -- if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

備🄵

This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

`turtle.`**ontimer**(*fun*, *t=0*)

參數

- **fun** -- a function with no arguments
- **t** -- a number >= 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()

turtle.**done**()

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Input methods

turtle.**textinput**(*title*, *prompt*)

參數

- **title** -- string（字串）
- **prompt** -- string（字串）

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

參數

- **title** -- string（字串）
- **prompt** -- string（字串）
- **default** -- number (optional)
- **minval** -- number (optional)
- **maxval** -- number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Settings and special methods

turtle.**mode**(*mode=None*)

參數

mode -- one of the strings "standard", "logo" or "world"

Set turtle mode ("standard", "logo" or "world") and perform reset. If mode is not given, current mode is returned.

Mode ”standard” is compatible with old *turtle*. Mode ”logo” is compatible with most Logo turtle graphics. Mode ”world” uses user-defined ”world coordinates”. **Attention**: in this mode angles appear distorted if `x/y` unit-ratio doesn't equal 1.

Mode	Initial turtle heading	positive angles
”standard”	to the right (east)	counterclockwise
”logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

`turtle.`**`colormode`**(*cmode=None*)

參數

`cmode` -- one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

`turtle.`**`getcanvas`**()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

`turtle.`**`getshapes`**()

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

`turtle.`**`register_shape`**(*name*, *shape=None*)

`turtle.`**`addshape`**(*name*, *shape=None*)

There are four different ways to call this function:

(1) *name* is the name of an image file (PNG, GIF, PGM, and PPM) and *shape* is `None`: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

> **備🄵**
>
> Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

(2) *name* is an arbitrary string and *shape* is the name of an image file (PNG, GIF, PGM, and PPM): Install the corresponding image shape.

```
>>> screen.register_shape("turtle", "turtle.gif")
```

> **備🄵**
>
> Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

(3) *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(4) *name* is an arbitrary string and *shape* is a (compound) `Shape` object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

在 3.14 版的變更: Added support for PNG, PGM, and PPM image formats. Both a shape name and an image file name can be specified.

`turtle.`**`turtles`**`()`

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`

Return the height of the turtle window.

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`

Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

`turtle.`**`bye`**`()`

Shut the turtlegraphics window.

`turtle.`**`exitonclick`**`()`

Bind `bye()` method to mouse clicks on the Screen.

If the value "using_IDLE" in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own mainloop is active also for the client script.

`turtle.`**`save`**`(`*filename*, *overwrite=False*`)`

Save the current turtle drawing (and turtles) as a PostScript file.

參數

- **`filename`** -- the path of the saved PostScript file

- **overwrite** -- if `False` and there already exists a file with the given filename, then the function will raise a `FileExistsError`. If it is `True`, the file will be overwritten.

```
>>> screen.save("my_drawing.ps")
>>> screen.save("my_drawing.ps", overwrite=True)
```

在 3.14 版被加入.

`turtle.`**`setup`**(*width=_CFG['width']*, *height=_CFG['height']*, *startx=_CFG['leftright']*, *starty=_CFG['topbottom']*)

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

參數

- **width** -- if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- **height** -- if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** -- if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if `None`, center window horizontally
- **starty** -- if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if `None`, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

`turtle.`**`title`**(*titlestring*)

參數

titlestring -- a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.10.8 Public classes

class `turtle.`**`RawTurtle`**(*canvas*)

class `turtle.`**`RawPen`**(*canvas*)

參數

canvas -- a `tkinter.Canvas`, a *ScrolledCanvas* or a *TurtleScreen*

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

class `turtle.`**`Turtle`**

Subclass of RawTurtle, has the same interface but draws on a default *Screen* object created automatically when needed for the first time.

class `turtle.`**`TurtleScreen`**(*cv*)

參數

cv -- 一個 `tkinter.Canvas`

Provides screen oriented methods like *bgcolor()* etc. that are described above.

class `turtle.`**`Screen`**

Subclass of TurtleScreen, with *four methods added*.

class turtle.**ScrolledCanvas**(*master*)

參數

master -- some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class turtle.**Shape**(*type_*, *data*)

參數

type_ -- one of the strings ”polygon”, ”image”, ”compound”

Data structure modeling shapes. The pair `(type_, data)` must follow this specification:

*type_*	*data*
”polygon”	a polygon-tuple, i.e. a tuple of pairs of coordinates
”image”	an image (in this form only used internally!)
”compound”	`None` (a compound shape has to be constructed using the *`addcomponent()`* method)

addcomponent(*poly*, *fill*, *outline=None*)

參數

- **poly** -- a polygon, i.e. a tuple of pairs of numbers
- **fill** -- a color the *poly* will be filled with
- **outline** -- a color for the poly's outline (if given)

例如：

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

請見*Compound shapes*。

class turtle.**Vec2D**(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

- `a + b` 向量加法
- `a - b` 向量🄵法
- `a * b` 🄵積
- `k * a` and `a * k` multiplication with scalar
- `abs(a)` a 的🄵對值
- `a.rotate(angle)` 旋轉

25.10.9 解釋

A turtle object draws on a screen object, and there a number of key classes in the turtle object-oriented interface that can be used to create them and relate them to each other.

A *`Turtle`* instance will automatically create a *`Screen`* instance if one is not already present.

`Turtle` is a subclass of *`RawTurtle`*, which *doesn't* automatically create a drawing surface - a *canvas* will need to be provided or created for it. The *canvas* can be a `tkinter.Canvas`, *`ScrolledCanvas`* or *`TurtleScreen`*.

TurtleScreen is the basic drawing surface for a turtle. *Screen* is a subclass of TurtleScreen, and includes *some additional methods* for managing its appearance (including size and title) and behaviour. TurtleScreen's constructor needs a tkinter.Canvas or a *ScrolledCanvas* as an argument.

The functional interface for turtle graphics uses the various methods of Turtle and TurtleScreen/Screen. Behind the scenes, a screen object is automatically created whenever a function derived from a Screen method is called. Similarly, a turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen, the object-oriented interface must be used.

25.10.10 Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling *help()* on methods or functions displays the docstrings:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.
```

(繼續下一頁)

(繼續上一頁)

```
    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

`turtle.`**`write_docstringdict`**(*filename='turtle_docstringdict'*)

參數
`filename` -- a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script *`filename`*`.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use *`turtle`* with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```
width = 0.5
height = 0.75
```

(繼續下一頁)

(繼續上一頁)

```
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the *Screen.setup* method.
- Line 5 and 6 correspond to the arguments of the method *Screen.screensize*.
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.
- If you want to use no fill color (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as *turtle*).
- The entries *exampleturtle* and *examplescreen* define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- *using_IDLE*: Set this to `True` if you regularly work with IDLE and its `-n` switch (”no subprocess”). This will prevent *exitonclick()* to enter the mainloop.

There can be a `turtle.cfg` file in the directory where *turtle* is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

25.10.11 `turtledemo` --- Demo scripts

The *turtledemo* package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The *turtledemo* package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.

- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are:

Name	描述	Features
bytedesign	complex classical turtle graphics pattern	*`tracer()`*, delay, *`update()`*
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, ontimer
colormixer	experiment with r, g, b	*`ondrag()`*
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	super minimalistic drawing program	*`onclick()`*
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	*`stamp()`*
planet_and_moon	simulation of gravitational system	compound shapes, *`Vec2D`*
rosette	a pattern from the wikipedia article on turtle graphics	*`clone()`*, *`undo()`*
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	*`clone()`*
two_canvases	simple design	turtles on two canvases
yinyang	another elementary example	*`circle()`*

Have fun!

25.10.12 Changes since Python 2.6

- The methods *`Turtle.tracer`*, *`Turtle.window_width`* and *`Turtle.window_height`* have been eliminated. Methods with these names and functionality are now available only as methods of *`Screen`*. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding *`TurtleScreen`*/*`Screen`* methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of *`begin_fill()`* and *`end_fill()`* have changed slightly: now every filling process must be completed with an `end_fill()` call.
- A method *`Turtle.filling`* has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

25.10.13 Changes since Python 3.0

- The *`Turtle`* methods *`shearfactor()`*, *`shapetransform()`* and *`get_shapepoly()`* have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. *`tiltangle()`* has been enhanced in functionality: it now can be used to get or set the tilt angle.
- The *`Screen`* method *`onkeypress()`* has been added as a complement to *`onkey()`*. As the latter binds actions to the key release event, an alias: *`onkeyrelease()`* was also added for it.

- The method *Screen.mainloop* has been added, so there is no longer a need to use the standalone *mainloop()* function when working with *Screen* and *Turtle* objects.
- Two input methods have been added: *Screen.textinput* and *Screen.numinput*. These pop up input dialogs and return strings and numbers respectively.

CHAPTER 26

開發工具

本章所描述的模組可以幫助你編寫軟體。例如*pydoc* 模組可以根據模組的🄵容生成文件；*doctest* 和*unittest* 模組則包含編寫單元測試的框架，這些測試程式碼會自動執行🄵驗證輸出結果是否正確。

本章節所描述的模組列表🄵：

26.1 `typing` --- 支援型🄵提示

在 3.5 版被加入.

原始碼： Lib/typing.py

備🄵

Python runtime 不🄵制要求函式與變數的型🄵🄵釋。他們可以被第三方工具使用，如：型🄵檢查器、IDE、linter 等。

此模組提供 runtime 型🄵提示支援。

動腦筋思考下面的函式：

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

函式 `surface_area_of_cube` 需要一個引數且預期是一個*float* 的實例，如 `edge_length: float` 所指出的型🄵提示。這個函式預期會回傳一個*str* 的實例，如 `-> str` 所指出的提示。

🄵管型🄵提示可以是簡單類🄵，像是*float* 或*str*，他們也可以變得更🄵🄵雜。模組*typing* 提供一組更高階的型🄵提示詞🄵。

新功能會頻繁的新增至 `typing` 模組中。typing_extensions 套件🄵這些新功能提供了 backport（向後移植的）版本，提供給舊版本的 Python 使用。

也參考

型🄵小抄 (Typing cheat sheet)
型🄴提示的快速預覽（發布於 mypy 的文件中）

mypy 文件的型🄵系統參考資料 (Type System Reference) 章節
Python 的加🄴型🄵系統是基於 PEPs 進行標准化，所以這個參照 (reference) 應該在多數 Python 型🄴檢查器中廣🄵使用。（某些部分依然是特定給 mypy 使用。）

Python 的🄵態型🄵 (Static Typing)
由社群編寫的跨平台型🄵檢查器文件 (type-checker-agnostic) 詳細描述加🄵型🄵系統的功能、實用的加🄵型🄵衍伸工具、以及加🄵型🄵的最佳實踐 (best practice)。

26.1.1 Python 型🄵系統的技術規范

關於 Python 型🄴系統標准的 (canonical)、最新的技術規範可以在Python 型🄴系統的技術規範找到。

26.1.2 型🄵🄵名

一個型🄵🄵名被定義來使用 `type` 陳述式，其建立了*TypeAliasType* 的實例。在這個範例中，`Vector` 及 `list[float]` 會被當作和🄵態型🄵檢查器一樣同等對待：

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

型🄵🄵名對於簡化🄵雜的型🄵簽名 (complex type signature) 非常好用。舉例來🄴：

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

`type` 陳述式是 Python 3.12 的新功能。🄵了向後相容性，型🄵🄵名可以透過簡單的賦值來建立：

```
Vector = list[float]
```

或是用*TypeAlias* 標記，讓它明確的表示這是一個型🄵🄵名，而非一般的變數賦值：

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

26.1.3 NewType

使用*NewType* 輔助工具 (helper) 建立獨特型🄵:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

若它是原本型🄵的子類🄵，🄵態型🄵檢查器會將其視🄵一個新的型🄵。這對於幫助🄵取邏輯性錯誤非常有用:

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

你依然可以在對於型🄵 `UserId` 的變數中執行所有 `int` 的操作。這讓你可以在預期接受 `int` 的地方傳遞一個 `UserId`，還能預防你意外使用無效的方法建立一個 `UserId`:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

注意這只會透過🄵態型🄵檢查器🄵制檢查。在 runtime 中，陳述式 (statement) `Derived = NewType('Derived', Base)` 會使 `Derived` 成🄵一個 callable（可呼叫物件），會立即回傳任何你傳遞的引數。這意味著 expression （運算式）`Derived(some_value)` 不會建立一個新的類🄵或過度引入原有的函式呼叫。

更精確地🄵，expression `some_value is Derived(some_value)` 在 runtime 永遠🄵 true。

這會無法建立一個 `Derived` 的子型🄵:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

無論如何，這有辦法基於'衍生的' `NewType` 建立一個*NewType*:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

以及針對 `ProUserId` 的型🄵檢查會如期運作。

更多細節請見 **PEP 484**。

> **備🄵**
>
> 請記得使用型🄵🄵名是宣告兩種型🄵是互相相等的。使用 `type Alias = Original` 則會讓🄵態型🄵檢查器在任何情🄵之下將 `Alias` 視🄵與 `Original` 完全相等。這當你想把🄵雜的型🄵簽名進行簡化時，非常好用。
>
> 相反的，`NewType` 宣告一個型🄵會是另外一種型🄵的子類🄵。使用 `Derived = NewType('Derived', Original)` 會使🄵態型🄵檢查器將 `Derived` 視🄵 `Original` 的子類🄵，也意味著一個型🄵🄵 `Original`

的值，不能被使用在任何預期接收到型🄵 `Derived` 的值的區域。這當你想用最小的 runtime 成本預防邏輯性錯誤而言，非常有用。

在 3.5.2 版被加入.

在 3.10 版的變更: 現在的 `NewType` 比起一個函式更像一個類🄵。因此，比起一般的函式，呼叫 `NewType` 需要額外的 runtime 成本。

在 3.11 版的變更: 呼叫 `NewType` 的效能已經恢復與 Python 3.9 相同的水准。

26.1.4 🄵釋 callable 物件

函式，或者是其他*callable* 物件，可以使用*collections.abc.Callable* 或以🄵用的*typing.Callable* 進行🄵釋。`Callable[[int], str]` 象徵🄵一個函式，可以接受一個型🄵🄵*int* 的引數，🄵回傳一個*str*。

舉例來🄵:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # 主體

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # 主體

async def on_update(value: str) -> None:
    ...  # 主體

callback: Callable[[str], Awaitable[None]] = on_update
```

使用下標語法 (subscription syntax) 時，必須使用到兩個值，分🄵🄵引述串列以及回傳類🄵。引數串列必須🄵一個型🄵串列：*ParamSpec*、*Concatenate* 或是一個🄴節號 (ellipsis, . . .)。回傳類🄵必🄵一個單一類🄵。

若🄴節號字面值 . . . 被當作引數串列給定，其指出一個具任何、任意參數列表的 callable 會被接受:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # 也 OK
```

`Callable` 不如有可變數量引數的函式、*overloaded functions*、或是僅限關鍵字參數的函式，可以表示🄵雜簽名。然而，這些簽名可以透過定義一個具有 `__call__()` 方法的*Protocol* 類🄴進行表示:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...
```

(繼續下一頁)

(繼續上一頁)

```
batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type because of
                         # different name and kind in the callback
```

Callable 物件可以取用其他 callable 當作引數使用，可以透過*ParamSpec* 指出他們的參數型🄵是個🄵獨立的。另外，如果這個 callable 從其他 callable 新增或🄵除引數時，將會使用到*Concatenate* 運算子。他們可以分🄵🄵用 `Callable[ParamSpecVariable, ReturnType]` 以及 `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` 的形式。

在 3.10 版的變更: `Callable` 現已支援*ParamSpec* 以及*Concatenate*。請參🄵 **PEP 612** 🄵讀詳細🄵容。

也參考

ParamSpec 以及*Concatenate* 的文件中，提供範例如何在 `Callable` 中使用。

26.1.5 泛型

因🄵關於物件的型🄵資訊留存在容器之🄵，且無法使用通用的方式進行🄵態推論 (statically inferred)，許多標准函式庫的容器類🄵支援以下標來表示容器🄵預期的元素。

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

泛型函式及類🄵可以使用型🄵參數語法 (type parameter syntax) 進行參數化 (parameterize)：

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic over the TypeVar "T"
    return l[0]
```

或是直接使用*TypeVar* 工廠 (factory)：

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar "U"
    return l[1]
```

在 3.12 版的變更: 在 Python 3.12 中，泛型的語法支援是全新功能。

26.1.6 🄵釋元組 (tuple)

在 Python 大多數的容器當中，加🄵型🄵系統認🄵容器🄵的所有元素會是相同型🄵。舉例來🄵：

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are meant to be ints
```

(繼續下一頁)

(繼續上一頁)

```
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

list 只接受一個型🄵引數，所以型🄵檢查器可能在上述 `y` 賦值 (assignment) 觸發錯誤。類似的範例，*Mapping* 只接受兩個型🄵引數：第一個引數指出 keys（鍵）的型🄵；第二個引數指出 values（值）的型🄵。

然而，與其他多數的 Python 容器不同，在慣用的 (idiomatic) Python 程式碼中，元組可以擁有不完全相同型🄵的元素是相當常見的。🄵此，元組在 Python 的加🄵型🄵系統中是個特例 (special-cased)。*tuple* 接受任何數量的型🄵引數：

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

🄵了標示一個元組可以🄵任意長度，且所有元素皆是相同型🄵 `T`，請使用🄵節號字面值 (literal ellipsis) `...`：`tuple[T, ...]`。🄵了標示一個空元組，請使用 `tuple[()]`。單純使用 `tuple` 作🄵🄵釋會與使用 `tuple[Any, ...]` 相等：

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

26.1.7 類🄵物件的型🄵

一個變數被🄵釋🄵 `C` 可以接受一個型🄵🄵 `C` 的值。相對的，一個變數備🄵解🄵 `type[C]` （或已🄵用的*typing.Type[C]*）可以接受本身🄵該類🄵的值 -- 具體來🄵，他可能會接受 `C` 的類🄵物件。舉例來🄵：

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

請記得 `type[C]` 是共變 (covariant) 的：

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...
```

(繼續下一頁)

(繼續上一頁)

```
def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)       # OK
make_new_user(ProUser)    # Also OK: ``type[ProUser]`` is a subtype of ``type[User]``
make_new_user(TeamUser)   # Still fine
make_new_user(User())     # Error: expected ``type[User]`` but got ``User``
make_new_user(int)        # Error: ``type[int]`` is not a subtype of ``type[User]``
```

type 僅有的合法參數是類🄵、*Any*、型🄵變數以及這些型🄵任意組合成的聯集。舉例來🄵:

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)  # OK
new_non_team_user(ProUser)    # OK
new_non_team_user(TeamUser)   # Error: ``type[TeamUser]`` is not a subtype
                              # of ``type[BasicUser | ProUser]``
new_non_team_user(User)       # Also an error
```

`type[Any]` 等價於*type*，其🄵 Python metaclass 階層結構 (hierachy)。

26.1.8 Annotating generators and coroutines

A generator can be annotated using the generic type *Generator[YieldType, SendType, ReturnType]*. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generic classes in the standard library, the `SendType` of *Generator* behaves contravariantly, not covariantly or invariantly.

The `SendType` and `ReturnType` parameters default to `None`:

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

It is also possible to set these types explicitly:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Simple generators that only ever yield values can also be annotated as having a return type of either *Iterable[YieldType]* or *Iterator[YieldType]*:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

Async generators are handled in a similar fashion, but don't expect a `ReturnType` type argument (*AsyncGenerator[YieldType, SendType]*). The `SendType` argument defaults to `None`, so the following definitions are equivalent:

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)

async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

As in the synchronous case, *AsyncIterable[YieldType]* and *AsyncIterator[YieldType]* are available as well:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Coroutines can be annotated using *Coroutine[YieldType, SendType, ReturnType]*. Generic arguments correspond to those of *Generator*, for example:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

26.1.9 使用者定義泛型型🄵

一個使用者定義的類🄵可以被定義成一個泛型類🄵。

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

這個語法指出類🄵 `LoggedVar` 透過一個單一的型🄵變數 `T` 進行參數化 (parameterised)。這使得 `T` 在類🄵中有效的成🄵型🄵。

泛型類🄵隱性繼承了*Generic*。🄵了相容 Python 3.11 及更早版本，也可以明確的繼承*Generic* 🄵指出是一個泛型類🄵：

```
from typing import TypeVar, Generic

T = TypeVar('T')
```

(繼續下一頁)

(繼續上一頁)

```
class LoggedVar(Generic[T]):
    ...
```

泛型類🄵有 `__class_getitem__()` 方法，其意味著可以在 runtime 進行參數化（如下述的 `LoggedVar[int]`）：

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

一個泛型型🄵可以有任意數量的型🄵變數。所有種類的 *TypeVar* 都可以作🄵泛型型🄵的參數：

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Generic 的每個型🄵變數引數必不相同。因此以下是無效的：

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

泛型類🄵亦可以繼承其他類🄵：

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

當繼承泛型類🄵時，部份的型🄵參數可固定：

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

在這種情🄵下 `MyDict` 有一個單一的參數 `T`。

若使用泛型類🄵🄵🄵有特指型🄵參數，則會將每個位置視🄵*Any*。在下列的範例中 `MyIterable` 不是泛型，但隱性繼承了 `Iterable[Any]`：

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

使用者定義的泛型型🄵🄵名也有支援。例如：

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

🄵了向後相容性，泛型型🄵🄵名可以透過簡單的賦值來建立：

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

在 3.7 版的變更: *Generic* 不再是一個自訂的 metaclass。

在 3.12 版的變更: 在版本 3.12 新增了泛型及型🄵🄵名的語法支援。在之前的版本中，泛型類🄵必須顯性繼承*Generic* 或是包含一個型🄵變數在基底類🄵 (base) 當中。

使用者定義的參數運算式 (parameter expression) 泛型一樣有支援，透過 `[**P]` 格式的參數規格變數來進行表示。對於上述作🄵參數規格變數的型🄵變數，將持續被 `typing` 模組視🄵一個特定的型🄵變數。對此，其中一個例外是一個型🄵列表可以替代*ParamSpec*：

```
>>> class Z[T, **P]: ...  # T 🄵 TypeVar; P 🄵 ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

具有*ParamSpec* 的泛型類🄵可以透過顯性繼承*Generic* 進行建立。在這種情🄵下，不需要使用 `**`：

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

另外一個*TypeVar* 以及*ParamSpec* 之間的差🄵是，基於美觀因素，只有一個參數規格變數的泛型可以接受如 `X[[Type1, Type2, ...]]` 以及 `X[Type1, Type2, ...]` 的參數列表。在🄵部中，後者會被轉🄵🄵前者，所以在下方的範例中🄵相等的：

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

請記得，具有*ParamSpec* 的泛型在某些情🄵下替🄵之後可能不會有正確的 `__parameters__`，因🄵參數規格主要還是用於🄵態型🄵檢查。

在 3.10 版的變更: *Generic* 現在可以透過參數運算式來進行參數化。詳細🄵容請見*ParamSpec* 以及 **PEP 612**。

一個使用者定義的泛型類🄵可以將 ABC 作🄵他們的基底類🄵，且不會有 metaclass 衝突。泛型的 metaclass 則不支援。參數化泛型的輸出將被存🄵快取，而在 `typing` 模組中多數的型🄵皆🄵*hashable* 且可以比較相等性。

26.1.10 `Any` 型🄵

Any 是一種特🄵的型🄵。一個🄵態型🄵檢查器會將每個型🄵視🄵可相容於*Any* 且*Any* 也可以相容於每個型🄵。

這意味著如果在一個🄵*Any* 的值上執行任何操作或呼叫方法是可行的，且可以賦值給任意變數：

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

請注意，當賦予型🄵🄵*Any* 的值更精確的型🄵時，將不會執行任何型🄵檢查。舉例來🄵，🄵態型🄵檢查器不會在 runtime 中，將 `a` 賦值給 `s` 的情🄵下回報錯誤，🄵管 `s` 是被宣告🄵型🄵*str* 🄵接收到*int* 的值！

另外，所有缺少回傳型🄵或參數型🄵的函式將會隱性預設🄵*Any*：

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

當你需要混和動態及🄵態的型🄵程式碼，這個行🄵允許*Any* 被當作一個緊急出口 *(escape hatch)* 使用。

Any 的行🄵對比*object* 的行🄵。與*Any* 相似，所有的型🄵會作🄵*object* 的子型🄵。然而，不像*Any*，反之不亦然：*object* 🄵不是一個其他型🄵的子型🄵。

這意味著當一個值的型🄵🄵*object* 時，型🄵檢查器會拒🄵幾乎所有的操作，🄵將賦予這個值到一個特定型🄵變數（或是當作回傳值使用）視🄵一個型🄵錯誤。舉例來🄵：

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")
```

(繼續下一頁)

(繼續上一頁)

```
# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

使用*object*，將指出在型🄵安全 (typesafe) 的習慣之下一個值可以🄵任意型🄵。使用*Any*，將指出這個值是個動態型🄵。

26.1.11 標稱 (nominal) 子型🄵 vs 結構子型🄵

最初 **PEP 484** 定義 Python 🄵態型🄵系統使用標懲子型🄵。這意味著只有 `A` 🄵 `B` 的子類🄵時，`A` 才被允許使用在預期是類🄵 `B` 出現的地方。

這個需求之前也被運用在抽象基底類🄵，例如*Iterable*。這種方式的問題在於，一個類🄵需要顯式的標記來支援他們，這🄴不符合 Python 風格，也不像一個常見的慣用動態型🄵 Python 程式碼。舉例來🄴，下列程式碼符合 **PEP 484**：

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 可以透過使用上方的程式碼，且在類🄵定義時不用顯式基底類🄵解🄵這個問題，讓 `Bucket` 被🄵態型🄵檢查器隱性認🄵是 `Sized` 以及 `Iterable[int]` 兩者的子型🄵。這就是🄵所周知的結構子型🄵（或是🄵態鴨子型🄵）：

```
from collections.abc import Iterator, Iterable

class Bucket:  # 注意：🄵有基底類🄵
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

而且，基於一個特🄵的型🄵*Protocol* 建立子型🄵時，使用者可以定義新的協定🄵充份發揮結構子型🄵的優勢（請見下方範例）。

26.1.12 模組🄵容

模組 `typing` 定義了下列的類🄵、函式以及裝飾器。

特🄵型🄵原語 (primitive)

特🄵型🄵

這些可以在🄵釋中做🄵型🄵。他們🄵不支援 `[]` 的下標使用。

`typing.`**`Any`**

特🄵型🄵，指出一個不受約束 (unconstrained) 的型🄵。

- 所有型🄵皆與*Any* 相容。
- *Any* 相容於所有型🄵。

在 3.11 版的變更: *Any* 可以作🄵一個基礎類🄵。這對於在任何地方使用鴨子型🄵或是高度動態的型🄵，避免型🄵檢查器的錯誤是非常有用的。

typing.**AnyStr**

一個不受約束的型🄵變數。

定義：

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` 是對於函式有用的，他可以接受*str* 或*bytes* 引數但不可以將此兩種混合。

舉例來🄵：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")    # OK, output has type 'str'
concat(b"foo", b"bar")  # OK, output has type 'bytes'
concat("foo", b"bar")   # Error, cannot mix str and bytes
```

請注意，🄵管他的名稱相近，`AnyStr` 與*Any* 型🄵無關，更不代表是「任何字串」的意思。尤其，`AnyStr` 與 `str | bytes` 兩者不同且具有不同的使用情境：

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18: Deprecated in favor of the new type parameter syntax. Use `class A[T: (str, bytes)]: ...` instead of importing `AnyStr`. See **PEP 695** for more details.

In Python 3.16, `AnyStr` will be removed from `typing.__all__`, and deprecation warnings will be emitted at runtime when it is accessed or imported from `typing`. `AnyStr` will be removed from `typing` in Python 3.18.

typing.**LiteralString**

特🄵型🄵，只包含文本字串。

任何文本字串都相容於 `LiteralString`，對於另一個 `LiteralString` 亦是如此。然而，若是一個型🄵僅🄵 `str` 的物件則不相容。一個字串若是透過組合多個 `LiteralString` 型🄵的物件建立，則此字串也可以視🄵 `LiteralString`。

舉例來🄵：

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` 對於敏感的 API 來🄵是有用的，其中任意的使用者🄵生的字串可能會🄵生問題。舉例來🄵，上面兩個案例中🄵生的型🄵檢查器錯誤是脆弱的且容易受到 SQL 注入攻擊。

更多細節請見 **PEP 675**。

在 3.11 版被加入.

typing.**Never**

typing.**NoReturn**

Never 和 NoReturn 表示底部型🄵 (bottom type)，🄵一個🄵有任何成員的型🄵。

它們可以被用來代表一個不會回傳的函式，像是*sys.exit()*：

```
from typing import Never  # 或 NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

或被用來定義一個不應被呼叫的函式，因🄵不會有有效的引數，、像是*assert_never()*：

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

Never 以及 NoReturn 在型🄵系統中具有相同的意義且🄵態型🄵檢查器會將兩者視🄵相等。

在 3.6.2 版被加入: 新增*NoReturn*。

在 3.11 版被加入: 新增*Never*。

typing.**Self**

特🄵型🄵，用來表示目前類🄵之🄵 (enclosed class)。

舉例來🄵:

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is "SubclassOfFoo"
```

這個🄵釋在語意上相等於下列🄵容，且形式更🄵簡潔:

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

一般來🄵，如果某個東西回傳 `self` 如上方的範例所示，你則應該使用 `Self` 做🄵回傳值的🄵釋。若 `Foo.return_self` 被🄵釋🄵回傳 `"Foo"`，則型🄵檢查器應該推論這個從 `SubclassOfFoo.return_self` 回傳的物件🄵 `Foo` 型🄵，而🄵非回傳 `SubclassOfFoo` 型🄵。

其他常見的使用案例包含：

- *classmethod* 被用來作🄵替代的建構函式 (constructor) 🄵回傳 `cls` 參數的實例。
- 🄵釋一個回傳自己的 `__enter__()` 方法。

當類🄵被子類🄵化時，若方法不保證回傳一個子類🄵的實例，你不應該使用 `Self` 作🄵回傳🄵釋：

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

更多細節請見 **PEP 673**。

在 3.11 版被加入.

typing.**TypeAlias**

做🄵明確宣告一個型🄵🄵名 的特🄵🄵釋。

舉例來🄵：

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` 在舊的 Python 版本中特🄵有用，其🄵釋🄵名可以用來進行傳遞參照 (forward reference)，因🄵對於型🄵檢查器來🄵，分辨這些🄵名與一般的變數賦值相當困難：

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

更多細節請見 **PEP 613**。

在 3.10 版被加入.

在 3.12 版之後被🄵用：*TypeAlias* 被🄵用，請改用 `type` 陳述式來建立 *TypeAliasType* 的實例，其自然可以支援傳遞參照的使用。請注意，雖然 *TypeAlias* 以及 *TypeAliasType* 提供相似的用途且具有相似的名稱，他們是不同的，且後者不是前者的型🄵。現在還🄵有移除 *TypeAlias* 的計畫，但鼓勵使用者們遷移 (migrate) 至 `type` 陳述式。

特🄵型式

這些在🄵釋中可以當作型🄵使用。他們全都支援 `[]` 的下標使用，但每個都具有獨特的語法。

class typing.**Union**

聯集型🄵；`Union[X, Y]` 與 `X | Y` 是相等的，且都意味著 X 或 Y 兩者其一。

🄵了定義聯集，例如可以使用 `Union[int, str]` 或是使用簡寫 (shorthand) `int | str`。使用這種簡寫是非常推薦的。詳細請看：

- 引數必須🄵型🄵且必須有至少一個。
- 聯集中的聯集會是扁平化的 (flattened)，舉例來🄵：

```
Union[Union[int, str], float] == Union[int, str, float]
```

However, this does not apply to unions referenced through a type alias, to avoid forcing evaluation of the underlying *TypeAliasType*:

```
type A = Union[int, str]
Union[A, float] != Union[int, str, float]
```

- 單一引數的聯集會消失不見，舉例來🄵：

```
Union[int] == int  # 實際上建構函式會回傳 int
```

- 多余的引數會被略過，舉例來🄵：

```
Union[int, str, int] == Union[int, str] == int | str
```

- 當比較聯集時，引數的順序會被忽略，舉例來🄵：

```
Union[int, str] == Union[str, int]
```

- 你不能建立 `Union` 的子類🄵或是實例。
- 你不能寫成 `Union[X][Y]`。

在 3.7 版的變更: 請勿在 runtime 中將顯性子類🄵從聯集中移除。

在 3.10 版的變更: 現在可以將聯集寫成 `X | Y`。請見聯集型🄵運算式。

在 3.14 版的變更: *types.UnionType* is now an alias for *Union*, and both `Union[int, str]` and `int | str` create instances of the same class. To check whether an object is a `Union` at runtime, use `isinstance(obj, Union)`. For compatibility with earlier versions of Python, use `get_origin(obj) is typing.Union or get_origin(obj) is types.UnionType`.

typing.**Optional**

`Optional[X]` 與 `X | None` 是相等的（或是 `Union[X, None]`）。

請注意，這與具有預設值的選擇性引數 (optional argument) 不是相同的概念。一個具有預設值的選擇性引數的型🄵🄵釋中不具有 `Optional` 限定符 (qualifier)，單純的因🄵它就是選擇性的。舉例來🄵：

```
def foo(arg: int = 0) -> None:
    ...
```

另一方面，如果一個顯性的值 `None` 是被允許的，不論引數是不是選擇性的，`Optional` 都適用。舉例來🄵：

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

在 3.10 版的變更: 現在可以將 Optional 寫成 `X | None`。請見聯集型🄵運算式。

typing.**Concatenate**

用於🄵釋高階函式的特🄵型式。

`Concatenate` 可以被用在可呼叫物件與*ParamSpec* 的接合 (conjunction) 🄵🄵釋一個高階的 Callable 物件可以新增、移除、轉🄵另一個 Callable 物件的參數。使用方法是依照這個格式 `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`。`Concatenate` 目前只在*Callable 物件*中第一個引數使用時有效。`Concatenate` 的最後一個參數必須🄵一個*ParamSpec* 或是🄵節號 (`...`)。

舉例來🄴，🄵釋一個🄵裝飾過後的函式提供*threading.Lock* 的裝飾器 `with_lock`，`Concatenate` 可以用於指出 `with_lock` 預期一個 Callable 物件，該物件可以接受 `Lock` 作🄵第一引數，🄵回傳一個具有不同型🄵簽名 Callable 物件。在這種情🄴下，*ParamSpec* 指出回傳的 Callable 物件的參數型🄵會依賴傳遞的 Callable 物件的參數型🄵：

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

在 3.10 版被加入.

也參考

- **PEP 612** -- 參數技術規範變數
- *ParamSpec*
- 🄵釋 *callable* 物件

typing.**Literal**

特殊型🄴格式，用於定義「文本型🄵 (literal type)」。

`Literal` 可以用於型🄵檢查器🄵指出🄵釋物件具有一個與提供的文本相同的值。

舉例來🄴：

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` 不可以進行子類🄵化。在 runtime 之中，任意的值是允許作🄵 `Literal[...]` 的型🄵引數，但型🄵檢查器可能會加🄵限制。更多有關文本型🄵的詳細資訊請看 **PEP 586**。

其他細節：

- 引數必須🄵文本值且必須有至少一個。

- 巢狀的 `Literal` 會是扁平化的 (flattened)，舉例來🄵：

```
assert Literal[Literal[1, 2], 3] == Literal[1, 2, 3]
```

However, this does not apply to `Literal` types referenced through a type alias, to avoid forcing evaluation of the underlying *TypeAliasType*:

```
type A = Literal[1, 2]
assert Literal[A, 3] != Literal[1, 2, 3]
```

- 多余的引數會被略過，舉例來🄵：

```
assert Literal[1, 2, 1] == Literal[1, 2]
```

- 當比較文本時，引數的順序會被忽略，舉例來🄵：

```
assert Literal[1, 2] == Literal[2, 1]
```

- 你不能建立 `Literal` 的子類🄵或是實例。
- 你不能寫成 `Literal[X][Y]`。

在 3.8 版被加入.

在 3.9.1 版的變更: `Literal` 現在可以🄵除重🄵 (de-deplicate) 的參數。`Literal` 物件的相等性比較不再依照相依性排序。`Literal` 物件現在會在相等性比較期間，若任一個其中的參數無法*hashable* 時，則會引發一個*TypeError* 例外。

typing.**ClassVar**

特殊型🄵建構，用來標記類🄵變數。

如同在 **PEP 526** 中的介紹，一個變數🄵解被包裝在 ClassVar 中時，會指出一個給定的屬性 (attribute) 意圖被當作類🄵變數使用，且不該被設定成該類🄵的實例。使用方法如下：

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # 類🄵變數
    damage: int = 10                     # 實例變數
```

ClassVar 只接受型🄵請不得使用下標。

ClassVar 🄵不代表該類🄵本身，而且不應該和*isinstance()* 或是*issubclass()* 一起使用。*ClassVar* 不會改變 Python runtime 的行🄵，但它可以被第三方的型🄵檢查器使用。舉例來🄵，一個型🄵檢查器可能會標記下方的程式碼🄵一個錯誤：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

在 3.5.3 版被加入.

在 3.13 版的變更: *ClassVar* can now be nested in *Final* and vice versa.

typing.**Final**

特殊型🄵建構，用來指出給型🄵檢查器的最終名稱。

最終名稱不可以在任何作用域 (scope) 中重新賦值。在類🄵作用域中宣告的最終名稱，不得在子類🄵中進行覆寫 (override)。

舉例來🄵：

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
```

(繼續下一頁)

(繼續上一頁)

```
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

這些屬性 (property) 不會在 runtime 時進行檢查。更多詳細資訊請看 **PEP 591**。

在 3.8 版被加入.

在 3.13 版的變更: *Final* can now be nested in *ClassVar* and vice versa.

typing.**Required**

特殊型🄵建構，用來標記一個*TypedDict* 鍵值是必須的。

主要用於 `total=False` 的 TypedDict。更多細節請見*TypedDict* 與 **PEP 655**。

在 3.11 版被加入.

typing.**NotRequired**

特殊型🄵建構，用來標記一個*TypedDict* 鍵值是可能消失的。

更多細節請見*TypedDict* 與 **PEP 655**。

在 3.11 版被加入.

typing.**ReadOnly**

特殊型🄵建構，用來標記一個*TypedDict* 的項目是唯讀的。

舉例來🄵:

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1999  # allowed
   m["title"] = "The Matrix"  # 型🄵檢查器錯誤
```

這些屬性 (property) 不會在 runtime 時進行檢查。

更多細節請見*TypedDict* 與 **PEP 705**。

在 3.13 版被加入.

typing.**Annotated**

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a `__metadata__` attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python's static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the *@no_type_check* decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using *isinstance()*).

Annotated[<type>, <metadata>]

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis:

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

The first argument to `Annotated` must be a valid type. Multiple metadata elements can be supplied as `Annotated` supports variadic arguments. The order of the metadata elements is preserved and matters for equality checks:

```
@dataclass
class ctype:
     kind: str

a1 = Annotated[int, ValueRange(3, 10), ctype("char")]
a2 = Annotated[int, ctype("char"), ValueRange(3, 10)]

assert a1 != a2  # 順序是有意義的
```

It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation:

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] == Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

However, this does not apply to `Annotated` types referenced through a type alias, to avoid forcing evaluation of the underlying *TypeAliasType*:

```
type From3To10[T] = Annotated[T, ValueRange(3, 10)]
assert Annotated[From3To10[int], ctype("char")] != Annotated[
   int, ValueRange(3, 10), ctype("char")
]
```

Duplicated metadata elements are not removed:

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

`Annotated` can be used with nested and generic aliases:

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V" the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

`Annotated` cannot be used with an unpacked *TypeVarTuple*:

```
type Variadic[*Ts] = Annotated[*Ts, Ann1] = Annotated[T1, T2, T3, ..., Ann1]  # 無效
```

where `T1`, `T2`, ... are *TypeVars*. This is invalid as only one type should be passed to Annotated.

By default, *get_type_hints()* strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved:

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}
```

At runtime, the metadata associated with an `Annotated` type can be retrieved via the `__metadata__` attribute:

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

If you want to retrieve the original type wrapped by `Annotated`, use the `__origin__` attribute:

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

Note that using *get_origin()* will return `Annotated` itself:

```
>>> get_origin(Password)
typing.Annotated
```

也參考

PEP 593 - Flexible function and variable annotations
The PEP introducing `Annotated` to the standard library.

在 3.9 版被加入.

`typing.`**`TypeIs`**

Special typing construct for marking user-defined type predicate functions.

`TypeIs` can be used to annotate the return type of a user-defined type predicate function. `TypeIs` only accepts a single type argument. At runtime, functions marked this way should return a boolean and take at least one positional argument.

`TypeIs` aims to benefit *type narrowing* -- a technique used by static type checkers to determine a more precise type of an expression within a program's code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a "type predicate":

```
def is_str(val: str | float):
    # "isinstance" type predicate
    if isinstance(val, str):
```

(繼續下一頁)

(繼續上一頁)

```
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Sometimes it would be convenient to use a user-defined boolean function as a type predicate. Such a function should use `TypeIs[...]` or *TypeGuard* as its return type to alert static type checkers to this intention. `TypeIs` usually has more intuitive behavior than `TypeGuard`, but it cannot be used when the input and output types are incompatible (e.g., `list[object]` to `list[int]`) or when the function does not return `True` for all instances of the narrowed type.

Using `-> TypeIs[NarrowedType]` tells the static type checker that for a given function:

1. 回傳值是一個布林值。
2. If the return value is `True`, the type of its argument is the intersection of the argument's original type and `NarrowedType`.
3. If the return value is `False`, the type of its argument is narrowed to exclude `NarrowedType`.

舉例來🄴:

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass

def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)

def run(arg: Child | Unrelated):
    if is_parent(arg):
        # Type of ``arg`` is narrowed to the intersection
        # of ``Parent`` and ``Child``, which is equivalent to
        # ``Child``.
        assert_type(arg, Child)
    else:
        # Type of ``arg`` is narrowed to exclude ``Parent``,
        # so only ``Unrelated`` is left.
        assert_type(arg, Unrelated)
```

The type inside `TypeIs` must be consistent with the type of the function's argument; if it is not, static type checkers will raise an error. An incorrectly written `TypeIs` function can lead to unsound behavior in the type system; it is the user's responsibility to write such functions in a type-safe manner.

If a `TypeIs` function is a class or instance method, then the type in `TypeIs` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeIs[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` is an instance of `TypeB`, and if it returns `False`, it is not an instance of `TypeB`.

`TypeIs` also works with type variables. For more information, see **PEP 742** (Narrowing types with `TypeIs`).

在 3.13 版被加入.

`typing.`**`TypeGuard`**

Special typing construct for marking user-defined type predicate functions.

Type predicate functions are user-defined functions that return whether their argument is an instance of a particular type. `TypeGuard` works similarly to *TypeIs*, but has subtly different effects on type checking behavior (see below).

Using `-> TypeGuard` tells the static type checker that for a given function:

1. 回傳值是一個布林值。
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

舉例來🄴:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

`TypeIs` 和 `TypeGuard` 在以下幾個方面有所不同:

- `TypeIs` requires the narrowed type to be a subtype of the input type, while `TypeGuard` does not. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant.
- When a `TypeGuard` function returns `True`, type checkers narrow the type of the variable to exactly the `TypeGuard` type. When a `TypeIs` function returns `True`, type checkers can infer a more precise type combining the previously known type of the variable with the `TypeIs` type. (Technically, this is known as an intersection type.)
- When a `TypeGuard` function returns `False`, type checkers cannot narrow the type of the variable at all. When a `TypeIs` function returns `False`, type checkers can narrow the type of the variable to exclude the `TypeIs` type.

在 3.10 版被加入.

`typing.`**`Unpack`**

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a *type variable tuple* is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of *`typing.TypeVarTuple`* and *`builtins.tuple`* types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn't be used in certain places:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-compatible
```

`Unpack` can also be used along with *`typing.TypedDict`* for typing `**kwargs` in a function signature:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** for more details on using `Unpack` for `**kwargs` typing.

在 3.11 版被加入.

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the `type` statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

class typing.**Generic**

Abstract base class for generic types.

A generic type is typically declared by adding a list of type parameters after the class name:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

This class can then be used as follows:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

Type variable.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases:

```
class Sequence[T]:  # T 是一個 TypeVar
    ...
```

This syntax can also be used to create bounded and constrained type variables:

```
class StrSequence[S: str]:  # S is a TypeVar with a `str` upper bound;
    ...                     # we can say that S is "bounded by `str`"


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar constrained to str or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so:

```
T = TypeVar('T')  # 可以是任何東西
S = TypeVar('S', bound=str)  # 可以是任何 str 的子型🄵
A = TypeVar('A', str, bytes)  # 必須是 str 或 bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See *Generic* for more information on generic types. Generic functions work as follows:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bounded*, *constrained*, or neither, but cannot be both bounded *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. By default, manually created type variables are invariant. See **PEP 484** and **PEP 695** for more details.

Bounded type variables and constrained type variables have different semantics in several important ways. Using a *bounded* type variable means that the `TypeVar` will be solved using the most specific type possible:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

The upper bound of a type variable can be a concrete type, abstract type (ABC or Protocol), or even a union of types:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be either str or bytes␣
↪in a function call, but not both
```

在 runtime `isinstance(x, T)` 會引發 *TypeError*。

__name__

The name of the type variable.

__covariant__

Whether the type var has been explicitly marked as covariant.

__contravariant__

Whether the type var has been explicitly marked as contravariant.

__infer_variance__

Whether the type variable's variance should be inferred by type checkers.

在 3.12 版被加入.

__bound__

The upper bound of the type variable, if any.

在 3.12 版的變更: For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_bound()

An *evaluate function* corresponding to the *__bound__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__bound__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

在 3.14 版被加入.

__constraints__

A tuple containing the constraints of the type variable, if any.

在 3.12 版的變更: For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_constraints()

An *evaluate function* corresponding to the *__constraints__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__constraints__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

在 3.14 版被加入.

__default__

The default value of the type variable, or *typing.NoDefault* if it has no default.

在 3.13 版被加入.

evaluate_default()

An *evaluate function* corresponding to the *__default__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__default__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

在 3.14 版被加入.

has_default()

Return whether or not the type variable has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

在 3.13 版被加入.

在 3.12 版的變更: Type variables can now be declared using the type parameter syntax introduced by **PEP 695**. The `infer_variance` parameter was added.

在 3.13 版的變更: 新增對預設值的支援。

class typing.**TypeVarTuple**(*name*, *, *default=typing.NoDefault*)

Type variable tuple. A specialized form of *type variable* that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (`*`) before the name:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the `TypeVarTuple` constructor:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

Note the use of the unpacking operator `*` in `tuple[T, *Ts]`. Conceptually, you can think of `Ts` as a tuple of type variables `(T1, T2, ...)`. `tuple[T, *Ts]` would then become `tuple[T, *(T1, T2, ...)]`,

which is equivalent to `tuple[T, T1, T2, ...]`. (Note that in older versions of Python, you might see this written using *Unpack* instead, as `Unpack[Ts]`.)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables:

```
x: Ts          # 無效
x: tuple[Ts]   # 無效
x: tuple[*Ts]  # 正確的做法
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()      # Totally fine
int_array_2d: Array[int, Height, Width] = Array()  # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters:

```
x: tuple[*Ts, *Ts]            # 無效
class Array[*Shape, *Shape]:  # 無效
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args`:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

See **PEP 646** for more details on type variable tuples.

__name__

The name of the type variable tuple.

__default__

The default value of the type variable tuple, or *typing.NoDefault* if it has no default.

在 3.13 版被加入.

evaluate_default()

An *evaluate function* corresponding to the *__default__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__default__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

在 3.14 版被加入.

has_default()

Return whether or not the type variable tuple has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

在 3.13 版被加入.

在 3.11 版被加入.

在 3.12 版的變更: Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695**.

在 3.13 版的變更: 新增對預設值的支援。

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*, *default=typing.NoDefault*)

Parameter specification variable. A specialized version of *type variables*.

In type parameter lists, parameter specifications can be declared with two asterisks (`**`):

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, `ParamSpec` objects can also be created as follows:

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable -- a pattern commonly found in higher order functions and decorators. They are only valid when used in `Concatenate`, or as the first argument to `Callable`, or as parameters for user-defined Generics. See *Generic* for more information on generic types.

For example, to add basic logging to a function, one can create a decorator `add_logging` to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without `ParamSpec`, the simplest way to annotate this previously was to use a *TypeVar* with upper bound `Callable[..., Any]`. However this causes two problems:

1. The type checker can't type check the `inner` function because `*args` and `**kwargs` have to be typed *Any*.

2. *cast()* may be required in the body of the `add_logging` decorator when returning the `inner` function, or the static type checker must be told to ignore the `return inner`.

args

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of *ParamSpecArgs* and *ParamSpecKwargs*.

__name__

The name of the parameter specification.

__default__

The default value of the parameter specification, or *typing.NoDefault* if it has no default.

在 3.13 版被加入.

evaluate_default()

An *evaluate function* corresponding to the *__default__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__default__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

在 3.14 版被加入.

has_default()

Return whether or not the parameter specification has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

在 3.13 版被加入.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to *TypeVar*. However the actual semantics of these keywords are yet to be decided.

在 3.10 版被加入.

在 3.12 版的變更: Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695**.

在 3.13 版的變更: 新增對預設值的支援。

備🄵

Only parameter specification variables defined in global scope can be pickled.

也參考

- **PEP 612** -- 參數技術規範變數
- *Concatenate*
- 🄵釋 *callable* 物件

typing.**ParamSpecArgs**

typing.**ParamSpecKwargs**

Arguments and keyword arguments attributes of a *ParamSpec*. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling *get_origin()* on either of these objects will return the original `ParamSpec`:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

在 3.10 版被加入.

class typing.**TypeAliasType**(*name*, *value*, *, *type_params=()*)

The type of type aliases created through the `type` statement.

舉例來🄵:

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

在 3.12 版被加入.

__name__

The name of the type alias:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

The module in which the type alias was defined:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias's value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
```

(繼續下一頁)

(繼續上一頁)

```
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

evaluate_value()

An *evaluate function* corresponding to the *__value__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__value__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format:

```
>>> type Alias = undefined
>>> Alias.__value__
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> from annotationlib import Format, call_evaluate_function
>>> Alias.evaluate_value(Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
```

在 3.14 版被加入.

Unpacking

Type aliases support star unpacking using the `*Alias` syntax. This is equivalent to using `Unpack[Alias]` directly:

```
>>> type Alias = tuple[int, str]
>>> type Unpacked = tuple[bool, *Alias]
>>> Unpacked.__value__
tuple[bool, typing.Unpack[Alias]]
```

在 3.14 版被加入.

Other special directives

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.**NamedTuple**

Typed version of *collections.namedtuple()*.

用法:

```
class Employee(NamedTuple):
    name: str
    id: int
```

這等價於:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
```

(繼續下一頁)

(繼續上一頁)

```
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *`namedtuple()`* API.)

`NamedTuple` subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic:

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

Backward-compatible usage:

```
# For creating a generic NamedTuple on Python 3.11
T = TypeVar("T")

class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

在 3.6 版的變更: Added support for **PEP 526** variable annotation syntax.

在 3.6.1 版的變更: Added support for default values, methods, and docstrings.

在 3.8 版的變更: The `_field_types` and `__annotations__` attributes are now regular dictionaries instead of instances of `OrderedDict`.

在 3.9 版的變更: Removed the `_field_types` attribute in favor of the more standard `__annotations__` attribute which has the same information.

在 3.11 版的變更: Added support for generic namedtuples.

在 3.14 版的變更: Using *`super()`* (and the `__class__` *closure variable*) in methods of `NamedTuple` subclasses is unsupported and causes a *`TypeError`*.

Deprecated since version 3.13, will be removed in version 3.15: The undocumented keyword argument syntax for creating NamedTuple classes (`NT = NamedTuple("NT", x=int)`) is deprecated, and will be disallowed in 3.15. Use the class-based syntax or the functional syntax instead.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a NamedTuple class, failing to pass a value to the 'fields' parameter (`NT = NamedTuple("NT")`) is deprecated. Passing `None` to the 'fields' parameter (`NT = NamedTuple("NT", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a NamedTuple class with 0 fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

class typing.**NewType**(*name*, *tp*)

Helper class to create low-overhead *distinct types*.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

用法:

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at runtime
```

__module__

The module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

在 3.5.2 版被加入.

在 3.10 版的變更: `NewType` is now a class rather than a function.

class typing.**Protocol**(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Such classes are primarily used with static type checkers that recognize structural subtyping (static duck-typing), for example:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

See **PEP 544** for more details. Protocol classes decorated with *runtime_checkable()* (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures. Protocol classes without this decorator cannot be used as the second argument to *isinstance()* or *issubclass()*.

Protocol classes can be generic, for example:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

在 3.8 版被加入.

@typing.**runtime_checkable**

Mark a protocol class as a runtime protocol.

Such a protocol can be used with *isinstance()* and *issubclass()*. This allows a simple-minded structural check, very similar to "one trick ponies" in *collections.abc* such as *Iterable*. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

This decorator raises *TypeError* when applied to a non-protocol class.

> **備🄵**
>
> `runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, *ssl.SSLObject* is a class, therefore it passes an *issubclass()* check against *Callable*. However, the `ssl.SSLObject.__init__` method exists only to raise a *TypeError* with a more informative message, therefore making it impossible to call (instantiate) *ssl.SSLObject*.

> **備🄵**
>
> An *isinstance()* check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as *hasattr()* calls for structural checks in performance-sensitive code.

在 3.8 版被加入.

在 3.12 版的變更: The internal implementation of *isinstance()* checks against runtime-checkable protocols now uses *inspect.getattr_static()* to look up attributes (previously, *hasattr()* was used). As a result, some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.

在 3.12 版的變更: The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on *isinstance()* checks comparing objects to the protocol. See What's new in Python 3.12 for more details.

class typing.**TypedDict**(*dict*)

Special construct to add type hints to a dictionary. At runtime it is a plain *dict*.

`TypedDict` declares a dictionary type that expects all of its instances to have a certain set of keys, where each key is associated with a value of a consistent type. This expectation is not checked at runtime but is only enforced by type checkers. Usage:

```
class Point2D(TypedDict):
    x: int
    y: int
```

(繼續下一頁)

(繼續上一頁)

```
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

An alternative way to create a `TypedDict` is by using function-call syntax. The second argument must be a literal *`dict`*:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

This functional syntax allows defining keys which are not valid identifiers, for example because they are keywords or contain hyphens, or when key names must not be mangled like regular private names:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

class Definition(TypedDict):
    __schema: str  # mangled to `_Definition__schema`

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
Definition = TypedDict('Definition', {'__schema': str})  # not mangled
```

By default, all keys must be present in a `TypedDict`. It is possible to mark individual keys as non-required using *`NotRequired`*:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# 替代語法
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

This means that a `Point2D` `TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False`:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# 替代語法
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D` `TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False` `TypedDict` can be marked as required using *`Required`*:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# 替代語法
```

(繼續下一頁)

(繼續上一頁)

```
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for *`Generic`*. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # 一個非 TypedDict 的類🄵

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # 引發 TypeError
```

A `TypedDict` can be generic:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from *`Generic`* explicitly:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see annotations-howto for more information on annotations best practices), *`__total__`*, *`__required_keys__`*, and *`__optional_keys__`*.

`__total__`

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
```

(繼續下一頁)

(繼續上一頁)

```
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys marked with *`NotRequired`*, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use *`__required_keys__`* and *`__optional_keys__`* for introspection.

`__required_keys__`

在 3.9 版被加入.

`__optional_keys__`

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return *`frozenset`* objects containing required and non-required keys, respectively.

Keys marked with *`Required`* will always appear in `__required_keys__` and keys marked with *`NotRequired`* will always appear in `__optional_keys__`.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total`:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

在 3.9 版被加入.

備🄵

If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

Support for *`ReadOnly`* is reflected in the following attributes:

`__readonly_keys__`

A *`frozenset`* containing the names of all read-only keys. Keys are read-only if they carry the *`ReadOnly`* qualifier.

在 3.13 版被加入.

`__mutable_keys__`

A *`frozenset`* containing the names of all mutable keys. Keys are mutable if they do not carry the *`ReadOnly`* qualifier.

在 3.13 版被加入.

See the TypedDict section in the typing documentation for more examples and detailed rules.

在 3.8 版被加入.

在 3.11 版的變更: Added support for marking individual keys as *Required* or *NotRequired*. See **PEP 655**.

在 3.11 版的變更: Added support for generic `TypedDict`s.

在 3.13 版的變更: Removed support for the keyword-argument method of creating `TypedDict`s.

在 3.13 版的變更: Support for the *ReadOnly* qualifier was added.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a TypedDict class, failing to pass a value to the 'fields' parameter (`TD = TypedDict("TD")`) is deprecated. Passing `None` to the 'fields' parameter (`TD = TypedDict("TD", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a TypedDict class with 0 fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.

協定

The following protocols are provided by the `typing` module. All are decorated with *@runtime_checkable*.

class typing.**SupportsAbs**

An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsBytes**

一個有抽象方法 `__bytes__` 的 ABC。

class typing.**SupportsComplex**

一個有抽象方法 `__complex__` 的 ABC。

class typing.**SupportsFloat**

一個有抽象方法 `__float__` 的 ABC。

class typing.**SupportsIndex**

一個有抽象方法 `__index__` 的 ABC。

在 3.8 版被加入.

class typing.**SupportsInt**

一個有抽象方法 `__int__` 的 ABC。

class typing.**SupportsRound**

An ABC with one abstract method `__round__` that is covariant in its return type.

ABCs and Protocols for working with I/O

class typing.**IO**[*AnyStr*]()

class typing.**TextIO**[*AnyStr*]()

class typing.**BinaryIO**[*AnyStr*]()

Generic class `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by *open()*. Please note that these classes are not protocols, and their interface is fairly broad.

The protocols *io.Reader* and *io.Writer* offer a simpler alternative for argument types, when only the `read()` or `write()` methods are accessed, respectively:

```
def read_and_write(reader: Reader[str], writer: Writer[bytes]):
    data = reader.read()
    writer.write(data.encode())
```

Also consider using *collections.abc.Iterable* for iterating over the lines of an input stream:

```
def read_config(stream: Iterable[str]):
    for line in stream:
        ...
```

函式與裝飾器

typing.**cast**(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

typing.**assert_type**(*val*, *typ*, /)

Ask a static type checker to confirm that *val* has an inferred type of *typ*.

At runtime this does nothing: it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

This function is useful for ensuring the type checker's understanding of a script is in line with the developer's intentions:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

在 3.11 版被加入.

typing.**assert_never**(*arg*, /)

Ask a static type checker to confirm that a line of code is unreachable.

舉例來🄵:

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an *int* or a *str*, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type *float*. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, *Never*, and nothing else.

At runtime, this throws an exception when called.

也參考

Unreachable Code and Exhaustiveness Checking has more information about exhaustiveness checking with static typing.

在 3.11 版被加入.

typing.**reveal_type**(*obj*, /)

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

This can be useful when you want to debug how your type checker handles a particular piece of code.

At runtime, this function prints the runtime type of its argument to *sys.stderr* and returns the argument unchanged (allowing the call to be used within an expression):

```
x = reveal_type(1)  # 印出 "Runtime type is int"
print(x)  # 印出 "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

在 3.11 版被加入.

@typing.**dataclass_transform**(**, eq_default=True, order_default=False, kw_only_default=False, frozen_default=False, field_specifiers=(), **kwargs*)

Decorator to mark an object as providing *dataclass*-like behavior.

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime "magic" that transforms a class in a similar way to *@dataclasses.dataclass*.

Example usage with a decorator function:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

On a base class:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

On a metaclass:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
```

(繼續下一頁)

(繼續上一頁)

```
    id: int
    name: str
```

The `CustomerModel` classes defined above will be treated by type checkers similarly to classes created with *`@dataclasses.dataclass`*. For example, type checkers will assume these classes have `__init__` methods that accept `id` and `name`.

The decorated class, metaclass, or function may accept the following bool arguments which type checkers will assume have the same effect as they would have on the *`@dataclasses.dataclass`* decorator: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, and `slots`. It must be possible for the value of these arguments (`True` or `False`) to be statically evaluated.

The arguments to the `dataclass_transform` decorator can be used to customize the default behaviors of the decorated class, metaclass, or function:

參數

- **`eq_default`** (`bool`) -- Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **`order_default`** (`bool`) -- Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`kw_only_default`** (`bool`) -- Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`frozen_default`** (`bool`) -- Indicates whether the `frozen` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.

  在 3.12 版被加入.
- **`field_specifiers`** (*`tuple[Callable[..., Any], ...]`*) -- Specifies a static list of supported classes or functions that describe fields, similar to *`dataclasses.field()`*. Defaults to `()`.
- **`**kwargs`** (*`Any`*) -- Arbitrary other keyword arguments are accepted in order to allow for possible future extensions.

Type checkers recognize the following optional parameters on field specifiers:

表格 1: **Recognised parameters for field specifiers**

Parameter name	Description
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

At runtime, this decorator records its arguments in the `__dataclass_transform__` attribute on the decorated object. It has no other runtime effect.

更多細節請見 **PEP 681**。

在 3.11 版被加入.

`@typing.`**`overload`**

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise *`NotImplementedError`*.

An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # 實際的實作在這邊
```

See **PEP 484** for more details and comparison with other typing semantics.

在 3.11 版的變更: Overloaded functions can now be introspected at runtime using *`get_overloads()`*.

`typing.`**`get_overloads`**(*func*)

Return a sequence of *`@overload`*-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for *`@overload`*, `get_overloads(process)` will return a sequence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` can be used for introspecting an overloaded function at runtime.

在 3.11 版被加入.

`typing.`**`clear_overloads`**()

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

在 3.11 版被加入.

`@typing.`**`final`**

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

舉例來🄵:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # 型🄵檢查器回報的錯誤
        ...
```

(繼續下一頁)

(繼續上一頁)

```
@final
class Leaf:
    ...
class Other(Leaf):  # 型🄴檢查器回報的錯誤
    ...
```

這些屬性 (property) 不會在 runtime 時進行檢查。更多詳細資訊請看 **PEP 591**。

在 3.8 版被加入.

在 3.11 版的變更: The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

`@typing.`**`no_type_check`**

Decorator to indicate that annotations are not type hints.

This works as a class or function *decorator*. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

`@typing.`**`no_type_check_decorator`**

Decorator to give another decorator the *`no_type_check()`* effect.

This wraps the decorator with something that wraps the decorated function in *`no_type_check()`*.

Deprecated since version 3.13, will be removed in version 3.15: No type checker ever added support for `@no_type_check_decorator`. It is therefore deprecated, and will be removed in Python 3.15.

`@typing.`**`override`**

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything. This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

舉例來🄴:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an object `obj` has been marked as an override. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

更多細節請見 **PEP 698**。

在 3.12 版被加入.

@typing.**type_check_only**

Decorator to mark a class or function as unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

Introspection helpers

typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary:

- Forward references encoded as string literals or *ForwardRef* objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*'s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.
- `None` is replaced with *types.NoneType*.
- If *@no_type_check* has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`'s base classes with those on `C` directly. This is done by traversing `C.__mro__` and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the *method resolution order* always take precedence over annotations on classes appearing later in the method resolution order.
- The function recursively replaces all occurrences of `Annotated[T, ...]` with `T`, unless *include_extras* is set to `True` (see *Annotated* for more information).

See also *annotationlib.get_annotations()*, a lower-level function that returns annotations more directly.

> **警示**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

> **備🄵**
>
> If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as *NameError*. For example, this can happen with imported *type aliases* that include forward references, or with names imported under *if TYPE_CHECKING*.

在 3.9 版的變更: 新增 `include_extras` 參數（如 **PEP 593** 中所述）。更多資訊請見*Annotated* 的文件。

在 3.11 版的變更: Previously, `Optional[t]` was added for function and method annotations if a default value equal to `None` was set. Now the annotation is returned unchanged.

typing.**get_origin**(*tp*)

Get the unsubscripted version of a type: for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or *collections* class, it will be normalized to the original class. If `X` is an instance of *ParamSpecArgs* or *ParamSpecKwargs*, return the underlying *ParamSpec*. Return `None` for unsupported objects.

舉例：

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

在 3.8 版被加入.

typing.**get_args**(*tp*)

Get type arguments with all substitutions performed: for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or *Literal* contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

舉例：

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

在 3.8 版被加入.

typing.**get_protocol_members**(*tp*)

Return the set of members defined in a *Protocol*.

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

Raise *TypeError* for arguments that are not Protocols.

在 3.13 版被加入.

typing.**is_protocol**(*tp*)

確定一個型🄵是否*Protocol*。

舉例來🄵：

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)     # => True
is_protocol(int)   # => False
```

在 3.13 版被加入.

typing.**is_typeddict**(*tp*)

Check if a type is a *TypedDict*.

舉例來🄵:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

在 3.10 版被加入.

class typing.**ForwardRef**

Class used for internal typing representation of string forward references.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. `ForwardRef` should not be instantiated by a user, but may be used by introspection tools.

> **備🄵**
>
> **PEP 585** generic types such as `list["SomeClass"]` will not be implicitly transformed into `list[ForwardRef("SomeClass")]` and thus will not automatically resolve to `list[SomeClass]`.

在 3.7.4 版被加入.

在 3.14 版的變更: This is now an alias for *annotationlib.ForwardRef*. Several undocumented behaviors of this class have been changed; for example, after a `ForwardRef` has been evaluated, the evaluated value is no longer cached.

typing.**evaluate_forward_ref**(*forward_ref*, *, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=annotationlib.Format.VALUE*)

Evaluate an *annotationlib.ForwardRef* as a *type hint*.

This is similar to calling *annotationlib.ForwardRef.evaluate()*, but unlike that method, `evaluate_forward_ref()` also recursively evaluates forward references nested within the type hint.

See the documentation for *annotationlib.ForwardRef.evaluate()* for the meaning of the *owner*, *globals*, *locals*, *type_params*, and *format* parameters.

> **警示**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

在 3.14 版被加入.

typing.**NoDefault**

A sentinel object used to indicate that a type parameter has no default value. For example:

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

在 3.13 版被加入.

常數

typing.**TYPE_CHECKING**

A special constant that is assumed to be `True` by 3rd party static type checkers. It's `False` at runtime.

A module which is expensive to import, and which only contain types used for typing annotations, can be safely imported inside an `if TYPE_CHECKING:` block. This prevents the module from actually being imported at runtime; annotations aren't eagerly evaluated (see **PEP 649**) so using undefined symbols in annotations is harmless--as long as you don't later examine them. Your static type analysis tool will set `TYPE_CHECKING` to `True` during static type analysis, which means the module will be imported and the types will be checked properly during such analysis.

用法:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: expensive_mod.SomeType) -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

If you occasionally need to examine type annotations at runtime which may contain undefined symbols, use *annotationlib.get_annotations()* with a `format` parameter of *annotationlib.Format.STRING* or *annotationlib.Format.FORWARDREF* to safely retrieve the annotations without raising *NameError*.

在 3.5.2 版被加入.

🄵用的🄵名

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the `typing` module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see **PEP 585**).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the `typing` module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

🄵建型🄵的🄵名

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

🄵用*dict* 的🄵名。

Note that to annotate arguments, it is preferred to use an abstract collection type such as *Mapping* rather than to use *dict* or `typing.Dict`.

在 3.9 版之後被🄵用: *builtins.dict* now supports subscripting (`[]`). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**List**(*list, MutableSequence[T]*)

🄵用*list* 的🄵名。

Note that to annotate arguments, it is preferred to use an abstract collection type such as *Sequence* or *Iterable* rather than to use *list* or `typing.List`.

在 3.9 版之後被🄵用: *builtins.list* now supports subscripting (`[]`). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Set**(*set, MutableSet[T]*)

🄵用*builtins.set* 的🄵名。

Note that to annotate arguments, it is preferred to use an abstract collection type such as *collections.abc.Set* rather than to use *set* or *typing.Set*.

在 3.9 版之後被🄵用: *builtins.set* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

🄵用*builtins.frozenset* 的🄵名。

在 3.9 版之後被🄵用: *builtins.frozenset* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

typing.**Tuple**

🄵用*tuple* 的🄵名。

tuple and Tuple are special-cased in the type system; see *🄵釋元組 (tuple)* for more details.

在 3.9 版之後被🄵用: *builtins.tuple* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Type**(*Generic[CT_co]*)

🄵用*type* 的🄵名。

See *類🄵物件的型🄵* for details on using *type* or typing.Type in type annotations.

在 3.5.2 版被加入.

在 3.9 版之後被🄵用: *builtins.type* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

collections 中型🄵的🄵名

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

🄵用*collections.defaultdict* 的🄵名。

在 3.5.2 版被加入.

在 3.9 版之後被🄵用: *collections.defaultdict* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

🄵用*collections.OrderedDict* 的🄵名。

在 3.7.2 版被加入.

在 3.9 版之後被🄵用: *collections.OrderedDict* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

🄵用*collections.ChainMap* 的🄵名。

在 3.6.1 版被加入.

在 3.9 版之後被🄵用: *collections.ChainMap* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

🄵用*collections.Counter* 的🄵名。

在 3.6.1 版被加入.

在 3.9 版之後被🄵用: *collections.Counter* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Deque**(*deque, MutableSequence[T]*)

🄵用*collections.deque* 的🄵名。

在 3.6.1 版被加入.

在 3.9 版之後被🄵用: *collections.deque* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

Aliases to other concrete types

class typing.**Pattern**

class typing.**Match**

Deprecated aliases corresponding to the return types from *re.compile()* and *re.match()*.

These types (and the corresponding functions) are generic over *AnyStr*. `Pattern` can be specialised as `Pattern[str]` or `Pattern[bytes]`; `Match` can be specialised as `Match[str]` or `Match[bytes]`.

在 3.9 版之後被🄵用: Classes `Pattern` and `Match` from *re* now support `[]`. See **PEP 585** and 泛型🄵名型🄵.

class typing.**Text**

🄵用*str* 的🄵名。

`Text` is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

在 3.5.2 版被加入.

在 3.11 版之後被🄵用: Python 2 is no longer supported, and most type checkers also no longer support type checking Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use *str* instead of `Text`.

collections.abc 中容器 ABC 的🄵名

class typing.**AbstractSet**(*Collection[T_co]*)

🄵用*collections.abc.Set* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Set* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

class typing.**ByteString**(*Sequence[int]*)

Deprecated alias to *collections.abc.ByteString*.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use *Buffer* or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both *bytes* and *bytearray*. However, since the ABC never had any methods, knowing that an object was an instance of `ByteString` never actually told you anything useful about the object. Other common buffer types such as *memoryview* were also never understood as subtypes of `ByteString` (either at runtime or by static type checkers).

See **PEP 688** for more details.

Deprecated since version 3.9, will be removed in version 3.17.

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)

🄵用*collections.abc.Collection* 的🄵名。

在 3.6 版被加入.

在 3.9 版之後被🄵用: *collections.abc.Collection* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Container**(*Generic[T_co]*)

🄵用*collections.abc.Container* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Container* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

🄵用*collections.abc.ItemsView* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.ItemsView* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)

🄵用*collections.abc.KeysView* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.KeysView* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Mapping**(*Collection[KT], Generic[KT, VT_co]*)

🄵用*collections.abc.Mapping* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Mapping* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**MappingView**(*Sized*)

🄵用*collections.abc.MappingView* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.MappingView* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**MutableMapping**(*Mapping[KT, VT]*)

🄵用*collections.abc.MutableMapping* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.MutableMapping* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**MutableSequence**(*Sequence[T]*)

🄵用*collections.abc.MutableSequence* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.MutableSequence* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**MutableSet**(*AbstractSet[T]*)

🄵用*collections.abc.MutableSet* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.MutableSet* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)

🄵用*collections.abc.Sequence* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Sequence* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)

🄵用*collections.abc.ValuesView* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.ValuesView* now supports subscripting ([]). See **PEP 585** and *泛型🄵名型🄵*.

collections.abc 中非同步 ABC 的🄵名

class typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

🄵用*collections.abc.Coroutine* 的🄵名。

See *Annotating generators and coroutines* for details on using *collections.abc.Coroutine* and `typing.Coroutine` in type annotations.

在 3.5.3 版被加入.

在 3.9 版之後被🄵用: *collections.abc.Coroutine* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

class typing.**AsyncGenerator**(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

🄵用*collections.abc.AsyncGenerator* 的🄵名。

See *Annotating generators and coroutines* for details on using *collections.abc.AsyncGenerator* and `typing.AsyncGenerator` in type annotations.

在 3.6.1 版被加入.

在 3.9 版之後被🄵用: *collections.abc.AsyncGenerator* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

在 3.13 版的變更: `SendType` 參數現在有預設值。

class typing.**AsyncIterable**(*Generic[T_co]*)

🄵用*collections.abc.AsyncIterable* 的🄵名。

在 3.5.2 版被加入.

在 3.9 版之後被🄵用: *collections.abc.AsyncIterable* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

🄵用*collections.abc.AsyncIterator* 的🄵名。

在 3.5.2 版被加入.

在 3.9 版之後被🄵用: *collections.abc.AsyncIterator* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

class typing.**Awaitable**(*Generic[T_co]*)

🄵用*collections.abc.Awaitable* 的🄵名。

在 3.5.2 版被加入.

在 3.9 版之後被🄵用: *collections.abc.Awaitable* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

Aliases to other ABCs in collections.abc

class typing.**Iterable**(*Generic[T_co]*)

🄵用*collections.abc.Iterable* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Iterable* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

class typing.**Iterator**(*Iterable[T_co]*)

🄵用*collections.abc.Iterator* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Iterator* now supports subscripting (`[]`). See **PEP 585** and 泛型🄵名型🄵.

typing.**Callable**

🄵用*collections.abc.Callable* 的🄵名。

See 🄵釋 *callable 物件* for details on how to use *collections.abc.Callable* and typing.Callable in type annotations.

在 3.9 版之後被🄵用: *collections.abc.Callable* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

在 3.10 版的變更: Callable 現已支援*ParamSpec* 以及*Concatenate*。請參🄵 **PEP 612** 🄵讀詳細🄵容。

class typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

🄵用*collections.abc.Generator* 的🄵名。

See *Annotating generators and coroutines* for details on using *collections.abc.Generator* and typing.Generator in type annotations.

在 3.9 版之後被🄵用: *collections.abc.Generator* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

在 3.13 版的變更: Default values for the send and return types were added.

class typing.**Hashable**

🄵用*collections.abc.Hashable* 的🄵名。

在 3.12 版之後被🄵用: 改🄵直接使用*collections.abc.Hashable*。

class typing.**Reversible**(*Iterable[T_co]*)

🄵用*collections.abc.Reversible* 的🄵名。

在 3.9 版之後被🄵用: *collections.abc.Reversible* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

class typing.**Sized**

🄵用*collections.abc.Sized* 的🄵名。

在 3.12 版之後被🄵用: 改🄵直接使用*collections.abc.Sized*。

contextlib ABC 的🄵名

class typing.**ContextManager**(*Generic[T_co, ExitT_co]*)

Deprecated alias to *contextlib.AbstractContextManager*.

The first type parameter, T_co, represents the type returned by the __enter__() method. The optional second type parameter, ExitT_co, which defaults to bool | None, represents the type returned by the __exit__() method.

在 3.5.4 版被加入.

在 3.9 版之後被🄵用: *contextlib.AbstractContextManager* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

在 3.13 版的變更: Added the optional second type parameter, ExitT_co.

class typing.**AsyncContextManager**(*Generic[T_co, AExitT_co]*)

Deprecated alias to *contextlib.AbstractAsyncContextManager*.

The first type parameter, T_co, represents the type returned by the __aenter__() method. The optional second type parameter, AExitT_co, which defaults to bool | None, represents the type returned by the __aexit__() method.

在 3.6.2 版被加入.

在 3.9 版之後被🄵用: *contextlib.AbstractAsyncContextManager* now supports subscripting ([]). See **PEP 585** and 泛型🄵名型🄵.

在 3.13 版的變更: Added the optional second type parameter, AExitT_co.

26.1.13 主要功能的🄵用時程表

Certain features in `typing` are deprecated and may be removed in a future version of Python. The following table summarizes major deprecations for your convenience. This is subject to change, and not all deprecations are listed.

Feature	🄵 用於	Projected removal	PEP/issue
`typing` versions of standard collections	3.9	Undecided (see 🄵用的🄵名 for more information)	**PEP 585**
typing.ByteString	3.9	3.17	gh-91896
typing.Text	3.11	Undecided	gh-92332
typing.Hashable 和 *typing.Sized*	3.12	Undecided	gh-94309
typing.TypeAlias	3.12	Undecided	**PEP 695**
@typing.no_type_check_decorator	3.13	3.15	gh-106309
typing.AnyStr	3.13	3.18	gh-105578

26.2 pydoc --- 文件🄵生器與🄵上幫助系統

原始碼： Lib/pydoc.py

`pydoc` 模組會自動從 Python 模組🄵生文件。文件可以以文字頁面的形式呈現在主控台上、提供給網頁🄵覽器，或儲存🄵 HTML 檔案。

對於模組、類🄵、函式和方法，所顯示的文件是從物件的🄵明字串（也就是物件的*__doc__* 屬性）以及遞🄵地從其可文件化成員的🄵明字串所取得。如果🄵有🄵明字串，`pydoc` 會嘗試從原始碼檔案中，類🄵、函式或方法定義正上方的🄵解區塊，或從模組的頂部取得描述（參見*inspect.getcomments()*）。

🄵建函式*help()* 會在互動式直譯器中呼叫🄵上幫助系統，它會使用 `pydoc` 在主控台上以文字形式🄵生文件。同樣的文字文件也可以在 Python 直譯器外部檢視，方法是在作業系統的命令提示字元中，將 **pydoc** 當作🄵本執行。例如，執行：

```
python -m pydoc sys
```

在 shell 提示字元下，將會顯示*sys* 模組的文件，其風格類似於 Unix **man** 命令所顯示的使用手🄵。**pydoc** 的引數可以是函式、模組或套件的名稱，或是一個指向模組中或套件中模組🄵的類🄵、方法或函式的點號參照 (dotted reference)。如果 **pydoc** 的引數看起來像一個路徑（也就是🄵，它包含你作業系統的路徑分隔符，例如 Unix 中的斜🄵），🄵且指向一個已存在的 Python 原始碼檔案，那🄵就會🄵該檔案🄵生文件。

> **備🄵**
>
> 🄵了找到物件及其文件，`pydoc` 會引入要文件化的模組。因此，任何在模組層級的程式碼都會在那個時候被執行。使用 `if __name__ == '__main__':` 的防護措施，可以確保程式碼只在檔案作🄵🄵本被呼叫時執行，而不會在被引入時就執行。

當在主控台印出輸出🄵容時，**pydoc** 會嘗試將輸出🄵容做分頁以便於🄵讀。如果 `MANPAGER` 或 `PAGER` 環境變數其中之一被設定，**pydoc** 將會使用其值作🄵分頁程式。當兩者都被設定時，會使用 `MANPAGER`。

在引數前指定一個 `-w` 旗標，將會使 HTML 文件被寫入目前目🄵的一個檔案中，而不是在主控台上顯示文字。

在引數前指定一個 `-k` 旗標，將會搜尋所有可用模組的摘要行 (synopsis lines)，尋找作🄵引數給出的關鍵字，其方式同樣類似於 Unix 的 **man** 命令。一個模組的摘要行是其🄵明字串的第一行。

你也可以使用 **pydoc** 在本機上🄵動一個 HTTP 伺服器，它會向來訪的網頁🄵覽器提供文件。**python -m pydoc -p 1234** 將會在 1234 埠🄵動一個 HTTP 伺服器，讓你可以用你偏好的網頁🄵覽器在 `http://localhost:1234/` 🄵覽文件。指定 `0` 作🄵通訊埠編號將會選擇一個任意未使用的通訊埠。

python -m pydoc -n <hostname> 將會🄵動伺服器🄵在給定的主機名稱上監聽。預設的主機名稱是'localhost'，但如果你希望伺服器能被其他機器連🄵，你可能會想改變伺服器回應的主機名稱。在開發期間，如果你想在容器🄵執行 pydoc，這點特🄵有用。

python -m pydoc -b 將會🄵動伺服器，🄵額外開🄵一個網頁🄵覽器到模組索引頁面。每個提供的頁面頂部都有一個導覽列，你可以在那🄵 *Get*（取得）個🄵項目的幫助、*Search*（搜尋）所有在其摘要行中包含關鍵字的模組，以及前往 *Module index*（模組索引）、*Topics*（主題）和 *Keywords*（關鍵字）頁面。

當 **pydoc** 🄵生文件時，它會使用目前的環境和路徑來定位模組。因此，呼叫 **pydoc spam** 所🄵生的文件，會完全對應到你🄵動 Python 直譯器🄵輸入 `import spam` 時所取得的模組版本。

核心模組的文件被假定存放在 `https://docs.python.org/X.Y/library/`，其中 `X` 和 `Y` 是 Python 直譯器的主版本號和次版本號。這可以透過將 `PYTHONDOCS` 環境變數設定🄵不同的 URL 或包含函式庫參考使用手🄵的本機目🄵來覆寫。

在 3.2 版的變更: 新增 `-b` 選項。

在 3.3 版的變更: 移除 `-g` 命令列選項。

在 3.4 版的變更: `pydoc` 現在使用 *inspect.signature()* 而不是 *inspect.getfullargspec()* 來從可呼叫物件中提取簽章資訊。

在 3.7 版的變更: 新增 `-n` 選項。

26.3 Python 開發模式

在 3.7 版被加入.

Python 開發模式引入了額外的 runtime 檢查，預設🄵用這些檢查的成本太高。如果程式碼正確，它不應比預設值更詳細；僅當偵測到問題時才會發出新警告。

可以使用 `-X dev` 命令列選項或將 `PYTHONDEVMODE` 環境變數設🄵 1 來🄵用它。

另請參🄵 Python 除錯建置。

26.3.1 Python 開發模式的影響

🄵用 Python 開發模式類似以下指令，但具有如下所述的附加效果：

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X faulthandler
```

Python 開發模式的效果：

- 新增 `default` 警告過濾器。以下警告會被顯示：
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  一般來🄵，上述警告會被預設的警告過濾器給過濾掉。

  它的行🄵就像使用 `-W default` 命令列選項一樣。

  使用 `-W error` 命令列選項或將 `PYTHONWARNINGS` 環境變數設🄵 `error` 會將警告視🄵錯誤。
- 在記憶體分配器上安裝除錯 hook（🄵🄵）以檢查：
  - 緩衝區下溢 (underflow)
  - 緩衝區溢位 (overflow)

  - 記憶體分配器 API 違規
  - GIL 的不安全使用

  請參🄵 `PyMem_SetupDebugHooks()` C 函式。

  它的行🄵就好像是將 `PYTHONMALLOC` 環境變數設定🄵 `debug` 一樣。

  若要🄵用 Python 開發模式而不在記憶體分配器上安裝偵錯 hook，請將 `PYTHONMALLOC` 環境變數設🄵 `default`。

- 在 Python 🄵動時呼叫 *`faulthandler.enable()`* 來🄵 *`SIGSEGV`*、*`SIGFPE`*、*`SIGABRT`*、*`SIGBUS`* 和 *`SIGILL`* 訊號安裝處理函式以在當機時傾印 (dump) Python 回溯 (traceback)。

  它的行🄵就像使用 `-X faulthandler` 命令列選項或將 `PYTHONFAULTHANDLER` 環境變數設定🄵 `1`。

- 🄵用 *asyncio 除錯模式*。例如 *`asyncio`* 會檢查未被等待的 (not awaited) 協程🄵記🄵 (log) 它們。

  它的行🄵就像將 `PYTHONASYNCIODEBUG` 環境變數設定🄵 1 一樣。

- 檢查字串編碼和解碼操作的 *encoding* 和 *errors* 引數。例如：*`open()`*、*`str.encode()`* 和 *`bytes.decode()`*。

  預設情🄵下，🄵了獲得最佳效能，僅在第一個編碼/解碼錯誤時檢查 *errors* 引數，🄵且有時會因🄵是空字串而忽略 *encoding* 引數。

- *`io.IOBase`* 解構函式會記🄵 `close()` 例外。

- 將 *`sys.flags`* 的 *`dev_mode`* 屬性設🄵 `True`。

Python 開發模式預設不會🄵用 *`tracemalloc`* 模組，因🄵（效能和記憶體的）開銷太大。🄵用 *`tracemalloc`* 模組可提供有關某些錯誤來源的附加資訊。例如 *`ResourceWarning`* 記🄵了分配資源之處的回溯、緩衝區溢位錯誤記🄵了分配記憶體區塊的回溯。

Python 開發模式不會防止 `-O` 命令列選項🄵除 `assert` 陳述式，也不會防止將 *`__debug__`* 設定🄵 `False`。

Python 開發模式只能在 Python 🄵動時🄵用。它的值可以從 *`sys.flags.dev_mode`* 讀取。

在 3.8 版的變更: *`io.IOBase`* 解構函式現在會記🄵 `close()` 例外。

在 3.9 版的變更: 現在會🄵字串編碼和解碼操作檢查 *encoding* 和 *errors* 引數。

26.3.2 ResourceWarning 范例

計算命令列中指定的文字檔案列數的🄵本範例：

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly

if __name__ == "__main__":
    main()
```

該🄵本不會明確關閉檔案。預設情🄵下，Python 不會發出任何警告。使用 README.txt 的範例，該檔案有 269 列：

```
$ python script.py README.txt
269
```

🄵用 Python 開發模式會顯示 *`ResourceWarning`* 警告：

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst' mode='r'␣
↪encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

此外，🄵用*tracemalloc* 會顯示檔案被開🄴的那一列：

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst' mode='r'␣
↪encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

修復方法是明確關閉該檔案。以下是使用情境管理器的範例：

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

不明確關閉資源可能會使資源開🄴的時間比預期的長得多；它可能會在退出 Python 時導致嚴重問題。在 CPython 中很糟糕，但在 PyPy 中更糟。明確關閉資源使應用程式更具確定性和可靠性。

26.3.3 檔案描述器的錯誤范例

顯示自身第一列的🄵本：

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

預設情🄵下，Python 不會發出任何警告：

```
$ python script.py
import os
```

Python 開發模式在最終化 (finalize) 檔案物件時顯示*ResourceWarning* 🄴記🄵"Bad file descriptor" 錯誤：

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py' mode='r'␣
↪encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8'>
Traceback (most recent call last):
```

(繼續下一頁)

(繼續上一頁)

```
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` 會關閉檔案描述器。當檔案物件最終化函式 (finalizer) 嘗試再次關閉檔案描述器時，它會失敗🄴出現 `Bad file descriptor` 錯誤。檔案描述器只能關閉一次。在最壞的情🄵下，將它關閉兩次可能會導致崩潰 (crash)（相關範例請參🄵 bpo-18748）。

修復方法是🄵除 `os.close(fp.fileno())` 那列，或使用 `closefd=False` 開🄵檔案。

26.4 `doctest` --- 測試互動式 Python 范例

原始碼： Lib/doctest.py

The *doctest* module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- 透過驗證測試檔案或測試物件中的互動式範例是否按預期運作來執行🄵歸測試。
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
```

(繼續下一頁)

(繼續上一頁)

```
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If you run `example.py` directly from the command line, *doctest* works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and *doctest* prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

That's all you need to know to start making productive use of *doctest*! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially

useful examples can be found in the standard test file `Lib/test/test_doctest/test_doctest.py`.

在 3.13 版被加入: Output is colorized by default and can be controlled using environment variables.

26.4.1 Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module `M` with:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

`doctest` then examines docstrings in module `M`.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to *`testmod()`*, or prohibit it by passing `verbose=False`. In either of those cases, *`sys.argv`* is not examined by *`testmod()`* (so passing `-v` or not has no effect).

There is also a command line shortcut for running *`testmod()`*, see section 命令列用法.

For more information on *`testmod()`*, see section 基礎 *API*.

26.4.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the *`testfile()`* function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *testmod()*, *testfile()* won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.

By default, *testfile()* looks for files in the calling module's directory. See section 基礎 *API* for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like *testmod()*, *testfile()*'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running *testfile()*, see section 命令列用法.

For more information on *testfile()*, see section 基礎 *API*.

26.4.3 命令列用法

可以從命令列以🄵本方式叫用*doctest* 模組:

```
python -m doctest [-v] [-o OPTION] [-f] file [file ...]
```

-v, --verbose

Detailed report of all examples tried is printed to standard output, along with assorted summaries at the end:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run *testmod()* on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

If the file name does not end with `.py`, `doctest` infers that it must be run with *testfile()* instead:

```
python -m doctest -v example.txt
```

-o, --option <option>

Option flags control various aspects of doctest's behavior, see section 可選旗標.

在 3.4 版被加入.

-f, --fail-fast

This is shorthand for `-o FAIL_FAST`.

在 3.4 版被加入.

26.4.4 運作方式

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched. In addition, there are cases when you want tests to be part of a module but not part of the help text, which requires that the tests not be included in the docstring. Doctest looks for a module-level variable called

`__test__` and uses it to locate other tests. If `M.__test__` exists, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `M.__test__.K`.

For example, place this block of code at the top of `example.py`:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module; if so, `doctest` searches them recursively for docstrings, which are then scanned for tests.

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

備🄵

`doctest` can only automatically discover classes and functions that are defined at the module level or inside other classes.

Since nested classes and functions only exist when an outer function is called, they cannot be discovered. Define them outside to make them visible.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # 🄵解會被忽略
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the

code output includes hard tabs, the only way the doctest can pass is if the *NORMALIZE_WHITESPACE* option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom *DocTestParser* class.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- The starting column doesn't matter:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time *doctest* finds a docstring to test, it uses a *shallow copy* of `M`'s globals, so that running tests doesn't change the module's real globals, and so that one test in `M` can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in `M`, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to *testmod()* or *testfile()* instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

簡單範例:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

That doctest succeeds if *ValueError* is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with *ValueError*) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether *ValueError* is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a `SyntaxError` that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some exceptions, Python displays the position of the error using ^ markers and tildes:

  ```
  >>> 1 + None
    File "<stdin>", line 1
      1 + None
      ~~^~~~~
  TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
  ```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

可選旗標

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*, and may be passed to the doctest command line interface via the `-o` option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**

By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When *DONT_ACCEPT_TRUE_FOR_1* is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

`doctest.`**`DONT_ACCEPT_BLANKLINE`**

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When *DONT_ACCEPT_BLANKLINE* is specified, this substitution is not allowed.

`doctest.`**`NORMALIZE_WHITESPACE`**

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. *NORMALIZE_WHITESPACE* is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

`doctest.`**`ELLIPSIS`**

When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don't match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *TypeError* is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
```

(繼續下一頁)

(繼續上一頁)

```
Traceback (most recent call last):
__main__.Exception: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

在 3.2 版的變更: *IGNORE_EXCEPTION_DETAIL* now also ignores any information relating to the module containing the exception under test.

doctest.**SKIP**

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

doctest.**COMPARISON_FLAGS**

A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.**REPORT_UDIFF**

When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.**REPORT_CDIFF**

When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.**REPORT_NDIFF**

When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.**REPORT_ONLY_FIRST_FAILURE**

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When *REPORT_ONLY_FIRST_FAILURE* is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.**FAIL_FAST**

When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

doctest.**REPORTING_FLAGS**

A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend *doctest* internals via subclassing:

doctest.**register_optionflag**(*name*)

Create a new option flag with a given name, and return the new flag's integer value. *register_optionflag()* can be used when subclassing *OutputChecker* or *DocTestRunner* to create new options that are supported by your subclasses. *register_optionflag()* should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive:             "#" "doctest:" directive_options
directive_options:     directive_option ("," directive_option)*
directive_option:      on_or_off directive_option_name
on_or_off:             "+" | "-"
directive_option_name: "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or - to disable it.

舉例來🄵，這個測試會通過：

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via - in a directive can be useful.

警告

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a set, Python doesn't guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{"spam", "eggs"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"spam", "eggs"}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0)  # certain to fail some of the time
7948648
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<C object at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for some floating-point calculations, and C libraries vary widely in quality here.

```
>>> 1000**0.1  # risky
1.9952623149688797
>>> round(1000**0.1, 9) # safer
1.995262315
>>> print(f'{1000**0.1:.4f}') # much safer
1.9953
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

26.4.5 基礎 API

The functions *testmod()* and *testfile()* provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections *Simple Usage: Checking Examples in Docstrings* and *Simple Usage: Checking Examples in a Text File*.

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).

- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like *dict.update()*: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in *sys.argv*.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value `0`) takes the bitwise OR of option flags. See section 可選旗標.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

回傳 `(failure_count, test_count)`。

Optional argument *name* gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using *doctest.master.summarize* in conjunction with *testmod()* continues to get output for objects with no tests. The *exclude_empty* argument to the newer *DocTestFinder* constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function *testfile()* above, except that *globs* defaults to `m.__dict__`.

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to `"NoName"`.

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function *testfile()* above.

26.4.6 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. *doctest* provides two functions that can be used to create *unittest* test suites from modules and text files containing doctests. To integrate with *unittest* test discovery, include a *load_tests* function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating *unittest.TestSuite* instances from text files and modules with doctests:

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convert doctest tests from one or more text files to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a *failureException* exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a *DocTest* object. The *setUp* function can access the test globals as the *globs* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a *DocTest* object. The *tearDown* function can access the test globals as the *globs* attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section 可選旗標. See function *set_unittest_reportflags()* below for a better way to set reporting options.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *optionflags=0*, *checker=None*)

Convert doctest tests for a module to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs each doctest in the module. Each docstring is run as a separate unit test. If any of the doctests fail, then the synthesized unit test fails, and a *unittest.TestCase.failureException* exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a docstring are skipped, then the

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is the module's `__dict__`.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above, but they are called for each docstring.

This function uses the same search technique as *testmod()*.

在 3.5 版的變更: *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of *unittest.TestCase*. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a *unittest.TestSuite* run instances of `DocTestCase`. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you're writing a *unittest* framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `unittest` to `doctest` test runners.

For this reason, *doctest* also supports a notion of `doctest` reporting flags specific to *unittest* support, via this function:

doctest.**set_unittest_reportflags**(*flags*)

Set the *doctest* reporting flags to use.

Argument *flags* takes the bitwise OR of option flags. See section 可選旗標. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module *unittest*: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`'s `unittest` reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, `doctest`'s `unittest` reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

26.4.7 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- *Example*: A single Python *statement*, paired with its expected output.
- *DocTest*: A collection of *Example*s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- *DocTestFinder*: Finds all docstrings in a given module, and uses a *DocTestParser* to create a *DocTest* from every docstring that contains interactive examples.
- *DocTestParser*: Creates a *DocTest* object from a string (such as an object's docstring).
- *DocTestRunner*: Executes the examples in a *DocTest*, and uses an *OutputChecker* to verify their output.
- *OutputChecker*: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:

```
                          list of:
+------+                  +---------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner-> results
+------+    |        ^     +---------+     |        ^     (printed)
            |        |     | Example |     |        |
            v        |     |   ...   |     v        |
           DocTestParser   | Example |   OutputChecker
                           +---------+
```

DocTest 物件

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of *Example* objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in *globs* after the test is run.

name

A string name identifying the *DocTest*. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this *DocTest* was extracted from; or `None` if the filename is unknown, or if the `DocTest` was not extracted from a file.

lineno

The line number within *filename* where this *DocTest* begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

Example 物件

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of *traceback.format_exception_only()*. *exc_msg* ends with a newline unless it's `None`. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s *optionflags*). By default, no options are set.

DocTestFinder 物件

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

A processing class used to extract the *DocTest*s that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTest*s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then *DocTestFinder.find()* will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then *DocTestFinder.find()* will include tests for objects with empty docstrings.

DocTestFinder 定義了以下方法：

find(*obj[, name][, module][, globs][, extraglobs]*)

Return a list of the *DocTest*s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned *DocTest*s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each *DocTest* is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each `DocTest`. If *globs* is not specified, then it defaults to the module's `__dict__`, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser 物件

class doctest.**DocTestParser**

A processing class used to extract interactive examples from a string, and use them to create a *DocTest* object.

DocTestParser 定義了以下方法：

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

Extract all doctest examples from the given string, and collect them into a *DocTest* object.

globs, *name*, *filename*, and *lineno* are attributes for the new `DocTest` object. See the documentation for *DocTest* for more information.

get_examples(*string*, *name='<string>'*)

Extract all doctest examples from the given string, and return them as a list of *Example* objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)

Divide the given string into examples and intervening text, and return them as a list of alternating *Example*s and strings. Line numbers for the `Example`s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

TestResults 物件

class doctest.**TestResults**(*failed*, *attempted*)

failed

失敗的測試數量。

attempted

嘗試的測試數量。

skipped

跳過的測試數量。

在 3.13 版被加入.

DocTestRunner 物件

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A processing class used to execute and verify the interactive examples in a *DocTest*.

The comparison between expected outputs and actual outputs is done by an *OutputChecker*. This comparison may be customized with a number of option flags; see section 可選旗標 for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of `OutputChecker` to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to *run()*; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods *report_start()*, *report_success()*, *report_unexpected_exception()*, and *report_failure()*.

The optional keyword argument *checker* specifies the *OutputChecker* object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the *DocTestRunner*'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section 可選旗標.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the *tries*, *failures* and *skips* attributes. The *run()* and *summarize()* methods return a *TestResults* instance.

DocTestRunner 定義了以下方法：

report_start(*out*, *test*, *example*)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_success(*out*, *test*, *example*, *got*)

Report that the given example ran successfully. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_failure(*out*, *test*, *example*, *got*)

Report that the given example failed. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by *sys.exc_info()*). *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

Run the examples in *test* (a *DocTest* object), and display the results using the writer function *out*. Return a *TestResults* instance.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the *DocTestRunner*'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize(*verbose=None*)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a *TestResults* instance.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the *DocTestRunner*'s verbosity is used.

DocTestParser 有以下屬性：

tries

嘗試的範例數量。

failures

失敗的範例數量。

skips

跳過的範例數量。

在 3.13 版被加入.

OutputChecker 物件

class doctest.**OutputChecker**

A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns `True` if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

OutputChecker 定義了以下方法：

check_output(*want*, *got*, *optionflags*)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section 可選旗標 for more information about option flags.

output_difference(*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

26.4.8 偵錯

Doctest 提供了幾種偵錯 doctest 範例的機制：

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite()* support the *debug()* method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace()* in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
```

(繼續下一頁)

(繼續上一頁)

```
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

doctest.**script_from_examples**(*s*)

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

doctest.**testsource**(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for *script_from_examples()* above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

doctest.**debug**(*module*, *name*, *pm=False*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function *testsource()* above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, *pdb*.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via *pdb.post_mortem()*, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate *exec()* call to *pdb.run()*.

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

Debug the doctests in a string.

This is like function *debug()* above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function *debug()* above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The *DebugRunner* class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially *DebugRunner*'s docstring (which is a doctest!) for more details:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A subclass of *DocTestRunner* that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an *UnexpectedException* exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a *DocTestFailure* exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for *DocTestRunner* in section *Advanced API*.

There are two exceptions that may be raised by *DebugRunner* instances:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

An exception raised by *DocTestRunner* to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure 定義了以下屬性:

DocTestFailure.**test**

The *DocTest* object that was being run when the example failed.

DocTestFailure.**example**

The *Example* that failed.

DocTestFailure.**got**

The example's actual output.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

An exception raised by *DocTestRunner* to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException 定義了以下屬性:

UnexpectedException.**test**

The *DocTest* object that was being run when the example failed.

UnexpectedException.**example**

The *Example* that failed.

UnexpectedException.**exc_info**

A tuple containing information about the unexpected exception, as returned by *sys.exc_info()*.

26.4.9 Soapbox

As mentioned in the introduction, *doctest* has grown to have three primary uses:

1. Checking examples in docstrings.
2. 回歸測試。
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned---it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my *doctest* examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using *testfile()* or *DocFileSuite()*. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a *__test__* dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

26.5 unittest --- 單元測試框架

原始碼： Lib/unittest/__init__.py

（假如你已經熟悉相關基礎的測試概念，你可能會希望跳過以下段落，直接參考*assert 方法清單*。）

unittest 原生的單元測試框架最初由 JUnit 開發，和其他程式語言相似有主要的單元測試框架。支援自動化測試，對測試分享安裝與關閉程式碼，集合所有匯總的測試，🄵且獨立各個測試報告框架。

unittest 用來作🄵實現支援一些重要的物件導向方法的概念：

test fixture
: 一個 *test fixture* 代表執行一個或多個測試所需要的准備，以及其他相關清理操作，例如可以是建立臨時性的或是代理用 (proxy) 資料庫、目🄵、或是🄵動一個伺服器程序。

test case（測試用例）
: 一個 *test case* 是一個獨立的單元測試。這是用來確認一個特定設定的輸入的特殊回饋。*unittest* 提供一個基礎類🄵，類🄵*TestCase*，可以用來建立一個新的測試條例。

test suite（測試套件）
: *test suite* 是一個搜集測試條例，測試套件，或是兩者皆有。它需要一起被執行🄵用來匯總測試。

test runner（測試執行器）
: *test runner* 是一個編排測試執行與提供結果給使用者的一個元件。執行器可以使用圖形化介面，文字介面或是回傳一個特🄵值用來標示出執行測試的結果。

也參考

***doctest* 模組**
: 另一個執行測試的模組，但使用不一樣的測試方法與規範。

Simple Smalltalk Testing: With Patterns
: Kent Beck 的原始論文討論使用*unittest* 這樣模式的測試框架。

pytest
: 第三方的單元測試框架，但在撰寫測試時使用更輕量的語法。例如：`assert func(10) == 42`。

The Python Testing Tools Taxonomy
: 一份詳細的 Python 測試工具列表，包含 functional testing 框架和 mock object 函式庫。

Testing in Python Mailing List
: 一個專門興趣的群組用來討論 Python 中的測試方式與測試工具。

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins, GitHub Actions, or AppVeyor.

26.5.1 簡單范例

unittest 模組提供一系列豐富的工具用來建構與執行測試。本節將展示這一系列工具中一部份，它們已能滿足大部份使用者需求。

這是一段簡短的🄵本用來測試 3 個字串方法：

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

(繼續下一頁)

(繼續上一頁)

```
    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

測試用例 (test case) 可以透過繼承*unittest.TestCase* 類🄵來建立。這🄵定義了三個獨立的物件方法，名稱皆以 `test` 開頭。這樣的命名方式能告知 test runner 哪些物件方法🄵定義的測試。

每個測試的關鍵🄵呼叫*assertEqual()* 來確認是否🄵期望的結果；*assertTrue()* 或是*assertFalse()* 用來驗證一個條件式；*assertRaises()* 用來驗證是否觸發一個特定的 exception。使用這些物件方法來取代 `assert` 陳述句，將能使 test runner 收集所有的測試結果🄵🄵生一個報表。

The *setUp()* and *tearDown()* methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section *Organizing test code*.

最後將顯示一個簡單的方法去執行測試*unittest.main()* 提供一個命令列介面測試🄵本。當透過命令列執行，輸出結果將會像是：

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

在測試時加入 `-v` 選項將指示*unittest.main()* 提高 verbosity 層級，🄵生以下的輸出：

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上的例子顯示大多數使用*unittest* 特徵足以滿足大多數日常測試的需求。接下來第一部分文件的剩余部分將繼續探索完整特徵設定。

在 3.11 版的變更: The behavior of returning a value from a test method (other than the default `None` value), is now deprecated.

26.5.2 命令列介面

單元測試模組可以透過命令列執行測試模組，物件甚至個🄵的測試方法：

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

你可以通過一個串列與任何模組名稱的組合，完全符合類🄵與方法的名稱。

測試模組可以根據檔案路徑指定：

```
python -m unittest tests/test_something.py
```

這允許你使用 shell 檔案名稱補完功能 (filename completion) 來指定測試模組。給定的檔案路徑必須亦能被當作模組 import。此路徑轉🄵🄵模組名稱的方式🄵移除'.py' 🄵將路徑分隔符 (path separator) 轉🄵成'.'。假如你的測試檔案無法被 import 成模組，你應該直接執行該測試檔案。

通過增加 -v 的旗標數，可以在你執行測試時得到更多細節（更高的 verbosity）：

```
python -m unittest -v test_module
```

若執行時不代任何引數，將執行*Test Discovery*（測試探索）：

```
python -m unittest
```

列出所有命令列選項：

```
python -m unittest -h
```

在 3.2 版的變更: 在早期的版本可以個🄵執行測試方法和不需要模組或是類🄵。

在 3.14 版被加入: Output is colorized by default and can be controlled using environment variables.

命令列模式選項

unittest 支援以下命令列選項：

-b, --buffer

Standard output 與 standard error stream 將在測試執行被緩衝 (buffer)。這些輸出在測試通過時被🄵🄵。若是測試錯誤或失則，這些輸出將會正常地被印出，🄵且被加入至錯誤訊息中。

-c, --catch

`Control-C` 測試執行過程中等待正確的測試結果🄵回報目前🄵止所有的測試結果。第二個 `Control-C` 🄵出一般例外*`KeyboardInterrupt`*。

參照*Signal Handling* 針對函式提供的功能。

-f, --failfast

在第一次錯誤或是失敗停止執行測試。

-k

Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match any of the given patterns are included.

Patterns that contain a wildcard character (`*`) are matched against the test name using *`fnmatch.fnmatchcase()`*; otherwise simple case-sensitive substring matching is used.

Patterns are matched against the fully qualified test method name as imported by the test loader.

For example, `-k foo` matches `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`.

--locals

透過 traceback 顯示本地變數。

--durations `N`

Show the N slowest test cases (N=0 for all).

在 3.2 版被加入: 增加命令列模式選項 `-b` 、`-c` 與 `-f`。

在 3.5 版被加入: 命令列選項 `--locals`。

在 3.7 版被加入: 命令列選項 `-k`。

在 3.12 版被加入: 命令列選項 `--durations`。

對執行所有的專案或是一個子集合測試，命令列模式可以可以被用來做測試探索。

26.5.3 Test Discovery（測試探索）

在 3.2 版被加入.

單元測試支援簡單的 test discovery（測試探索）。🄵了相容於測試探索，所有的測試檔案都要是模組或是套件，🄵能從專案的最上層目🄵中 import（代表它們的檔案名稱必須是有效的 identifiers）。

Test discovery（測試探索）實作在*TestLoader.discover()*，但也可以被用於命令列模式。基本的命令列模式用法如下：

```
cd project_directory
python -m unittest discover
```

備🄵

`python -m unittest` 作🄵捷徑，其功能相當於 `python -m unittest discover`。假如你想傳遞引數至探索測試的話，一定要明確地加入 `discover` 子指令。

`discover` 子命令有以下幾個選項：

-v, --verbose
詳細 (verbose) 輸出

-s, --start-directory directory
開始尋找的資料夾（預設🄵 .）

-p, --pattern pattern
匹配測試檔案的模式（預設🄵 `test*.py`）

-t, --top-level-directory directory
專案的最高階層目🄵（預設🄵開始的資料夾）

-s, *-p*, 和*-t* 選項依照傳遞位置作🄵引數排序順序。以下兩個命令列被視🄵等價：

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

警示

Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the *load_tests protocol*.

在 3.4 版的變更: Test discovery supports *namespace packages*.

在 3.11 版的變更: Test discovery dropped the *namespace packages* support. It has been broken since Python 3.7. Start directory and its subdirectories containing tests must be regular package that have `__init__.py` file.

If the start directory is the dotted name of the package, the ancestor packages can be namespace packages.

在 3.14 版的變更: Test discovery supports *namespace package* as start directory again. To avoid scanning directories unrelated to Python, tests are not searched in subdirectories that do not contain `__init__.py`.

26.5.4 Organizing test code

The basic building blocks of unit testing are *test cases* --- single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the *assert* methods* provided by the `TestCase` base class. If the test fails, an exception will be raised with an explanatory message, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

> **備🄵**
>
> The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
```

(繼續下一頁)

(繼續上一頁)

```
    def tearDown(self):
        self.widget.dispose()
```

If *setUp()* succeeded, *tearDown()* will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a *test fixture*. A new TestCase instance is created as a unique test fixture used to execute each individual test method. Thus *setUp()*, *tearDown()*, and __init__() will be called once per test.

It is recommended that you use TestCase implementations to group tests together according to the features they test. *unittest* provides a mechanism for this: the *test suite*, represented by *unittest*'s *TestSuite* class. In most cases, calling *unittest.main()* will do the right thing and collect all the module's test cases for you and execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

26.5.5 Re-using old test code

Some users will find that they have existing test code that they would like to run from *unittest*, without converting every old test function to a *TestCase* subclass.

For this reason, *unittest* provides a *FunctionTestCase* class. This subclass of *TestCase* can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

> **備🄵**
>
> Even though *FunctionTestCase* can be used to quickly convert an existing test base over to a *unittest*-based system, this approach is not recommended. Taking the time to set up proper *TestCase* subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the *doctest* module. If so, *doctest* provides a DocTestSuite class that can automatically build *unittest.TestSuite* instances from the existing *doctest*-based tests.

26.5.6 Skipping tests and expected failures

在 3.1 版被加入.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a *TestResult*.

Skipping a test is simply a matter of using the *skip()* *decorator* or one of its conditional variants, calling *TestCase.skipTest()* within a *setUp()* or test method, or raising *SkipTest* directly.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not supported in this library␣
↪version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ... skipped 'external resource␣
↪not available'
test_windows_support (__main__.MyTestCase.test_windows_support) ... skipped 'requires Windows
↪'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Classes can be skipped just like methods:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the *expectedFailure()* decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls *skip()* on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators and exception implement test skipping and expected failures:

`@unittest.`**`skip`**`(`*reason*`)`

Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.

`@unittest.`**`skipIf`**`(`*condition*`, `*reason*`)`

Skip the decorated test if *condition* is true.

`@unittest.`**`skipUnless`**`(`*condition*`, `*reason*`)`

Skip the decorated test unless *condition* is true.

`@unittest.`**`expectedFailure`**

Mark the test as an expected failure or error. If the test fails or errors in the test function itself (rather than in one of the *test fixture* methods) then it will be considered a success. If the test passes, it will be considered a failure.

`exception` `unittest.`**`SkipTest`**`(`*reason*`)`

This exception is raised to skip a test.

Usually you can use *TestCase.skipTest()* or one of the skipping decorators instead of raising this directly.

Skipped tests will not have *setUp()* or *tearDown()* run around them. Skipped classes will not have *setUpClass()* or *tearDownClass()* run. Skipped modules will not have *setUpModule()* or *tearDownModule()* run.

26.5.7 Distinguishing test iterations using subtests

在 3.4 版被加入.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the *subTest()* context manager.

舉例來🄵，以下測試：

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
```

(繼續下一頁)

(繼續上一頁)

```
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

會有以下輸出：

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of `i` wouldn't be displayed:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.5.8 類🄵與函式

This section describes in depth the API of *unittest*.

Test cases

class unittest.**TestCase**(*methodName='runTest'*)

Instances of the *TestCase* class represent the logical test units in the *unittest* universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of *TestCase* will run a single base method: the method named *methodName*. In most uses of *TestCase*, you will neither change the *methodName* nor reimplement the default `runTest()` method.

在 3.2 版的變更: *TestCase* can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with *TestCase* from the interactive interpreter.

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

setUp()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than *AssertionError* or *SkipTest*, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than *AssertionError* or *SkipTest*, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the *setUp()* succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

更多細節請見*Class and Module Fixtures*。

在 3.2 版被加入.

tearDownClass()

A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

更多細節請見*Class and Module Fixtures*。

在 3.2 版被加入.

run(*result=None*)

Run the test, collecting the result into the *TestResult* object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the *defaultTestResult()* method) and used. The result object is returned to *run()*'s caller.

The same effect may be had by simply calling the *TestCase* instance.

在 3.3 版的變更: Previous versions of `run` did not return the result. Neither did calling an instance.

skipTest(*reason*)

Calling this during a test method or *setUp()* skips the current test. See *Skipping tests and expected failures* for more information.

在 3.1 版被加入.

subTest(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

更多資訊請見*Distinguishing test iterations using subtests*。

在 3.4 版被加入.

debug()

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

方法	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2
`assertIsSubclass(a, b)`	`issubclass(a, b)`	3.14
`assertNotIsSubclass(a, b)`	`not issubclass(a, b)`	3.14

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the *msg* keyword argument can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the *list of type-specific methods*).

在 3.1 版的變更: Added the automatic calling of type-specific equality function.

在 3.2 版的變更: `assertMultiLineEqual()` added as the default type equality function for comparing strings.

assertNotEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

Test that *first* and *second* are (or are not) the same object.

在 3.1 版被加入.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

Test that *expr* is (or is not) `None`.

在 3.1 版被加入.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

Test that *member* is (or is not) in *container*.

在 3.1 版被加入.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls)`.

在 3.2 版被加入.

assertIsSubclass(*cls*, *superclass*, *msg=None*)

assertNotIsSubclass(*cls*, *superclass*, *msg=None*)

Test that *cls* is (or is not) a subclass of *superclass* (which can be a class or a tuple of classes, as supported by `issubclass()`). To check for the exact type, use `assertIs(cls, superclass)`.

在 3.14 版被加入.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

方法	Checks that	New in
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` 會引發 *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` 會引發 *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*	3.4
`assertNoLogs(logger, level)`	**The `with` block does not log on** *logger* with minimum *level*	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, *, *msg=None*)

Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if *exception* is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, *assertRaises()* accepts the additional keyword argument *msg*.

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

在 3.1 版的變更: 新增*assertRaises()* 可作🄵情境管理器使用的能力。

在 3.2 版的變更: 新增 `exception` 屬性。

在 3.3 版的變更: 新增作🄵情境管理器使用時的 *msg* 引數。

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, *, *msg=None*)

Like *assertRaises()* but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

或是:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

在 3.1 版被加入: 以 `assertRaisesRegexp` 🄵名新增。

在 3.2 版的變更: 重新命名🄵*assertRaisesRegex()*。

在 3.3 版的變更: 新增作🄵情境管理器使用時的 *msg* 引數。

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, *, *msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to *assertWarns()*. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, *assertWarns()* accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

在 3.2 版被加入.

在 3.3 版的變更: 新增作🄵情境管理器使用時的 *msg* 引數。

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, *, *msg=None*)

Like *assertWarns()* but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Example:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

或是:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

在 3.2 版被加入.

在 3.3 版的變更: 新增作🄵情境管理器使用時的 *msg* 引數。

assertLogs(*logger=None*, *level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a *logging.Logger* object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages that were not blocked by a non-propagating descendent logger.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or *logging.ERROR*). The default is *logging.INFO*.

The test passes if at least one message emitted inside the `with` block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records

A list of *logging.LogRecord* objects of the matching log messages.

output

A list of *str* objects with the formatted output of matching messages.

範例:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

在 3.4 版被加入.

assertNoLogs(*logger=None*, *level=None*)

A context manager to test that no messages are logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

Unlike `assertLogs()`, nothing will be returned by the context manager.

在 3.10 版被加入.

There are also other methods used to perform more specific checks, such as:

方法	Checks that	New in
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* and *b* have the same elements in the same number, regardless of their order.	3.2
`assertStartsWith(a, b)`	`a.startswith(b)`	3.14
`assertNotStartsWith(a, b)`	`not a.startswith(b)`	3.14
`assertEndsWith(a, b)`	`a.endswith(b)`	3.14
`assertNotEndsWith(a, b)`	`not a.endswith(b)`	3.14
`assertHasAttr(a, b)`	`hastattr(a, b)`	3.14
`assertNotHasAttr(a, b)`	`not hastattr(a, b)`	3.14

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the `round()` function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a `TypeError`.

在 3.2 版的變更: `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)

assertGreaterEqual(*first*, *second*, *msg=None*)

assertLess(*first*, *second*, *msg=None*)

assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

在 3.1 版被加入.

assertRegex(*text*, *regex*, *msg=None*)

assertNotRegex(*text*, *regex*, *msg=None*)

Test that a *regex* search matches (or does not match) *text*. In case of failure, the error message will include the pattern and the *text* (or the pattern and the part of *text* that unexpectedly matched). *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*.

在 3.1 版被加入: 以 `assertRegexpMatches` 🄵名新增。

在 3.2 版的變更: `assertRegexpMatches()` 方法已重新命名🄵*assertRegex()*。

在 3.2 版被加入: *assertNotRegex()*。

assertCountEqual(*first*, *second*, *msg=None*)

Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

在 3.2 版被加入.

assertStartsWith(*s*, *prefix*, *msg=None*)

assertNotStartsWith(*s*, *prefix*, *msg=None*)

Test that the Unicode or byte string *s* starts (or does not start) with a *prefix*. *prefix* can also be a tuple of strings to try.

在 3.14 版被加入.

assertEndsWith(*s*, *suffix*, *msg=None*)

assertNotEndsWith(*s*, *suffix*, *msg=None*)

Test that the Unicode or byte string *s* ends (or does not end) with a *suffix*. *suffix* can also be a tuple of strings to try.

在 3.14 版被加入.

assertHasAttr(*obj*, *name*, *msg=None*)

assertNotHasAttr(*obj*, *name*, *msg=None*)

Test that the object *obj* has (or has not) an attribute *name*.

在 3.14 版被加入.

The *assertEqual()* method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using *addTypeEqualityFunc()*:

addTypeEqualityFunc(*typeobj*, *function*)

Registers a type-specific method called by *assertEqual()* to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as *assertEqual()* does. It must raise *self.failureException(msg)* when inequality between the first two parameters is detected -- possibly providing useful information and explaining the inequalities in details in the error message.

在 3.1 版被加入.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

方法	Used to compare	New in
`assertMultiLineEqual(a, b)`	字串	3.1
`assertSequenceEqual(a, b)`	序列	3.1
`assertListEqual(a, b)`	串列	3.1
`assertTupleEqual(a, b)`	元組	3.1
`assertSetEqual(a, b)`	集合或凍結集合	3.1
`assertDictEqual(a, b)`	字典	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

在 3.1 版被加入.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and `assertTupleEqual()`.

在 3.1 版被加入.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

在 3.1 版被加入.

assertSetEqual(*first*, *second*, *msg=None*)

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of *first* or *second* does not have a `set.difference()` method.

在 3.1 版被加入.

assertDictEqual(*first*, *second*, *msg=None*)

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

在 3.1 版被加入.

Finally the `TestCase` provides the following methods and attributes:

fail(*msg=None*)

Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is `AssertionError`.

longMessage

This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

在 3.1 版被加入.

maxDiff

This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are *assertSequenceEqual()* (including all the sequence comparison methods that delegate to it), *assertDictEqual()* and *assertMultiLineEqual()*.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

在 3.2 版被加入.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()

Return the number of tests represented by this test object. For *TestCase* instances, this will always be `1`.

defaultTestResult()

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the *run()* method).

For *TestCase* instances, this will always be an instance of *TestResult*; subclasses of *TestCase* should override this as necessary.

id()

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()

Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or `None`.

在 3.1 版的變更: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the *TextTestResult* in Python 3.2.

addCleanup(*function*, /, **args*, ***kwargs*)

Add a function to be called after *tearDown()* to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addCleanup()* when they are added.

If *setUp()* fails, meaning that *tearDown()* is not called, then any cleanup functions added will still be called.

在 3.1 版被加入.

enterContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by *addCleanup()* and return the result of the `__enter__()` method.

在 3.11 版被加入.

doCleanups()

This method is called unconditionally after *tearDown()*, or after *setUp()* if *setUp()* raises an exception.

It is responsible for calling all the cleanup functions added by *addCleanup()*. If you need cleanup functions to be called *prior* to *tearDown()* then you can call *doCleanups()* yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

在 3.1 版被加入.

classmethod addClassCleanup(*function*, */*, **args*, ***kwargs*)

Add a function to be called after *tearDownClass()* to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addClassCleanup()* when they are added.

If *setUpClass()* fails, meaning that *tearDownClass()* is not called, then any cleanup functions added will still be called.

在 3.8 版被加入.

classmethod enterClassContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by *addClassCleanup()* and return the result of the `__enter__()` method.

在 3.11 版被加入.

classmethod doClassCleanups()

This method is called unconditionally after *tearDownClass()*, or after *setUpClass()* if *setUpClass()* raises an exception.

It is responsible for calling all the cleanup functions added by *addClassCleanup()*. If you need cleanup functions to be called *prior* to *tearDownClass()* then you can call *doClassCleanups()* yourself.

doClassCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

在 3.8 版被加入.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)

This class provides an API similar to *TestCase* and also accepts coroutines as test functions.

在 3.8 版被加入.

loop_factory

The *loop_factory* passed to *asyncio.Runner*. Override in subclasses with *asyncio.EventLoop* to avoid using the asyncio policy system.

在 3.13 版被加入.

async asyncSetUp()

Method called to prepare the test fixture. This is called after `setUp()`. This is called immediately before calling the test method; other than *AssertionError* or *SkipTest*, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

async asyncTearDown()

Method called immediately after the test method has been called and the result recorded. This is called before `tearDown()`. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than *AssertionError* or *SkipTest*, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the *asyncSetUp()* succeeds, regardless of the outcome of the test method. The default implementation does nothing.

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)

This method accepts a coroutine that can be used as a cleanup function.

async enterAsyncContext(*cm*)

Enter the supplied *asynchronous context manager*. If successful, also add its `__aexit__()` method as a cleanup function by *`addAsyncCleanup()`* and return the result of the `__aenter__()` method.

在 3.11 版被加入.

run(*result=None*)

Sets up a new event loop to run the test, collecting the result into the *`TestResult`* object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to *`run()`*'s caller. At the end of the test all the tasks in the event loop are cancelled.

An example illustrating the order:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

After running the test, `events` would contain `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

This class implements the portion of the *`TestCase`* interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a *`unittest`*-based test framework.

Grouping tests

class unittest.**TestSuite**(*tests=()*)

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a *TestSuite* instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

TestSuite objects behave much like *TestCase* objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to *TestSuite* instances:

addTest(*test*)

Add a *TestCase* or *TestSuite* to the suite.

addTests(*tests*)

Add all the tests from an iterable of *TestCase* and *TestSuite* instances to this test suite.

This is equivalent to iterating over *tests*, calling *addTest()* for each element.

TestSuite shares the following methods with *TestCase*:

run(*result*)

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike *TestCase.run()*, *TestSuite.run()* requires the result object to be passed in.

debug()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a *TestSuite* are always accessed by iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before *TestSuite.run()* must be the same for each call iteration. After *TestSuite.run()*, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

在 3.2 版的變更: In earlier versions the *TestSuite* accessed tests directly rather than through iteration, so overriding `__iter__()` wasn't sufficient for providing tests.

在 3.4 版的變更: In earlier versions the *TestSuite* held references to each *TestCase* after *TestSuite.run()*. Subclasses can restore that behavior by overriding `TestSuite._removeTestAtIndex()`.

In the typical usage of a *TestSuite* object, the *run()* method is invoked by a `TestRunner` rather than by the end-user test harness.

Loading and running tests

class unittest.**TestLoader**

The *TestLoader* class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the *unittest* module provides an instance that can be shared as *unittest.defaultTestLoader*. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

在 3.5 版被加入.

TestLoader objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)

Return a suite of all test cases contained in the *TestCase*-derived `testCaseClass`.

A test case instance is created for each method named by *getTestCaseNames()*. By default these are the method names beginning with `test`. If *getTestCaseNames()* returns no methods, but the `runTest()` method is implemented, a single test case is created for that method instead.

loadTestsFromModule(*module*, *, *pattern=None*)

Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from *TestCase* and creates an instance of the class for each test method defined for the class.

> **備🄵**
>
> While using a hierarchy of *TestCase*-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the *load_tests protocol*. The *pattern* argument is passed as the third argument to `load_tests`.

在 3.2 版的變更: Support for `load_tests` added.

在 3.5 版的變更: Support for a keyword-only argument *pattern* has been added.

在 3.12 版的變更: The undocumented and unofficial *use_load_tests* parameter has been removed.

loadTestsFromName(*name*, *module=None*)

Return a suite of all test cases given a string specifier.

The specifier *name* is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a *TestSuite* instance, or a callable object which returns a *TestCase* or *TestSuite* instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module `SampleTests` containing a *TestCase*-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

在 3.5 版的變更: If an *ImportError* or *AttributeError* occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar to *loadTestsFromName()*, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then *top_level_dir* must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves.

top_level_dir is stored internally, and used as a default to any nested calls to `discover()`. That is, if a package's `load_tests` calls `loader.discover()`, it does not need to pass this argument.

start_dir can be a dotted module name as well as a directory.

在 3.2 版被加入.

在 3.4 版的變更: Modules that raise `SkipTest` on import are recorded as skips, not errors.

start_dir can be a *namespace packages*.

Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

在 3.5 版的變更: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

在 3.11 版的變更: *start_dir* can not be a *namespace packages*. It has been broken since Python 3.7, and Python 3.11 officially removes it.

在 3.13 版的變更: *top_level_dir* is only stored for the duration of *discover* call.

在 3.14 版的變更: *start_dir* can once again be a *namespace package*.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

testMethodPrefix

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*` methods.

suiteClass

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the *TestSuite* class.

This affects all the `loadTestsFrom*` methods.

testNamePatterns

List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-k` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using *fnmatch.fnmatchcase()*, so unlike patterns passed to the `-k` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*` methods.

在 3.7 版被加入.

class unittest.**TestResult**

This class is used to compile information about which tests have succeeded and which have failed.

A *TestResult* object stores the results of a set of tests. The *TestCase* and *TestSuite* classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of *unittest* may want access to the *TestResult* object generated by running a set of tests for reporting purposes; a *TestResult* instance is returned by the `TestRunner.run()` method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the *assert* methods*.

skipped

A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

在 3.1 版被加入.

expectedFailures

A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure or error of the test case.

unexpectedSuccesses

A list containing *TestCase* instances that were marked as expected failures, but succeeded.

collectedDurations

A list containing 2-tuples of test case names and floats representing the elapsed time of each test which was run.

在 3.12 版被加入.

shouldStop

Set to `True` when the execution of tests should stop by *stop()*.

testsRun

The total number of tests run so far.

buffer

If set to true, `sys.stdout` and `sys.stderr` will be buffered in between *startTest()* and *stopTest()* being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

在 3.2 版被加入.

failfast

If set to true *stop()* will be called on the first failure or error, halting the test run.

在 3.2 版被加入.

tb_locals

If set to true then local variables will be shown in tracebacks.

在 3.5 版被加入.

wasSuccessful()

Return `True` if all tests run so far have passed, otherwise returns `False`.

在 3.4 版的變更: Returns `False` if there were any *unexpectedSuccesses* from tests marked with the *expectedFailure()* decorator.

stop()

This method can be called to signal that the set of tests being run should be aborted by setting the *shouldStop* attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the *TextTestRunner* class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the *TestResult* class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)

Called when the test case *test* is about to be run.

stopTest(*test*)

Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()

Called once before any tests are executed.

在 3.1 版被加入.

stopTestRun()

Called once after all tests are executed.

在 3.1 版被加入.

addError(*test*, *err*)

Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)

Called when the test case *test* signals a failure. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *failures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)

Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)

Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's *skipped* attribute.

addExpectedFailure(*test*, *err*)

Called when the test case *test* fails or errors, but was marked with the *expectedFailure()* decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *expectedFailures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)

Called when the test case *test* was marked with the *expectedFailure()* decorator, but succeeded.

The default implementation appends the test to the instance's *unexpectedSuccesses* attribute.

addSubTest(*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom *TestCase* instance describing the subtest.

If *outcome* is *None*, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

在 3.4 版被加入.

addDuration(*test*, *elapsed*)

Called when the test case finishes. *elapsed* is the time represented in seconds, and it includes the execution of cleanup functions.

在 3.12 版被加入.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*, *, *durations=None*)

A concrete implementation of *TestResult* used by the *TextTestRunner*. Subclasses should accept `**kwargs` to ensure compatibility as the interface changes.

在 3.2 版被加入.

在 3.12 版的變更: 新增 *durations* 關鍵字參數。

unittest.**defaultTestLoader**

Instance of the *TestLoader* class intended to be shared. If no customization of the *TestLoader* is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*, *durations=None*)

A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, *sys.stderr* is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept `**kwargs` as the interface to construct runners changes when features are added to unittest.

By default this runner shows *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning* and *ImportWarning* even if they are *ignored by default*. This behavior can be overridden using Python's `-Wd` or `-Wa` options (see Warning control) and leaving *warnings* to `None`.

在 3.2 版的變更: 新增 *warnings* 參數。

在 3.2 版的變更: The default stream is set to *sys.stderr* at instantiation time rather than import time.

在 3.5 版的變更: 新增 *tb_locals* 參數。

在 3.12 版的變更: 新增 *durations* 參數。

_makeResult()

This method returns the instance of `TestResult` used by *run()*. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to *TextTestResult* if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)

This method is the main public interface to the `TextTestRunner`. This method takes a *TestSuite* or *TestCase* instance. A *TestResult* is created by calling *_makeResult()* and the test(s) are run and the results printed to stdout.

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of *sys.argv* are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls *sys.exit()* with an exit code indicating success (0) or failure (1) of the tests run. An exit code of 5 indicates that no tests were run or skipped.

The *testLoader* argument has to be a *TestLoader* instance, and defaults to *defaultTestLoader*.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling *sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name *command-line options*.

The *warnings* argument specifies the *warning filter* that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` returns an object with the `result` attribute that contains the result of the tests run as a *unittest.TestResult*.

在 3.1 版的變更: 新增 *exit* 參數。

在 3.2 版的變更: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

在 3.4 版的變更: The *defaultTest* parameter was changed to also accept an iterable of test names.

load_tests 協定

在 3.2 版被加入.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by *`TestLoader.loadTestsFromModule()`* with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

它應該回傳一個*`TestSuite`*。

loader is the instance of *`TestLoader`* doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of *`TestCase`* classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling *`TestLoader.discover()`*, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a *`TestSuite`* representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

在 3.5 版的變更: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

26.5.9 Class and Module Fixtures

Class and module level fixtures are implemented in *`TestSuite`*. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a *`TestCase`*) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass 和 tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in *`TestCase`* are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a *`SkipTest`* exception then the class will be reported as having been skipped instead of as an error.

setUpModule 和 tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a *`SkipTest`* exception then the module will be reported as having been skipped instead of as an error.

To add cleanup code that must be run even in the case of an exception, use `addModuleCleanup`:

`unittest.`**`addModuleCleanup`**(*function*, */*, **args*, ***kwargs*)

Add a function to be called after *`tearDownModule()`* to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *`addModuleCleanup()`* when they are added.

If *`setUpModule()`* fails, meaning that *`tearDownModule()`* is not called, then any cleanup functions added will still be called.

在 3.8 版被加入.

unittest.**enterModuleContext**(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addModuleCleanup()` and return the result of the `__enter__()` method.

在 3.11 版被加入.

unittest.**doModuleCleanups**()

This function is called unconditionally after `tearDownModule()`, or after `setUpModule()` if `setUpModule()` raises an exception.

It is responsible for calling all the cleanup functions added by `addModuleCleanup()`. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call `doModuleCleanups()` yourself.

`doModuleCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

在 3.8 版被加入.

26.5.10 Signal Handling

在 3.2 版被加入.

The `-c/--catch` command-line option to unittest, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn't* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the `removeHandler()` decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.**installHandler**()

Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have `stop()` called.

unittest.**registerResult**(*result*)

Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

unittest.**removeResult**(*result*)

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

unittest.**removeHandler**(*function=None*)

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.6 `unittest.mock` —mock 物件函式庫

在 3.3 版被加入.

原始碼： Lib/unittest/mock.py

---

unittest.mock 在 Python 中是一個用於進行測試的函式庫。它允許你用 mock 物件在測試中替🄵部分系統，🄵判定它們是如何被使用的。

unittest.mock 提供了一個以*Mock* 🄵核心的類🄵，去除在測試中建立大量 stubs 的需求。在執行動作之後，你可以判定哪些 method （方法） / 屬性被使用，以及有哪些引數被呼叫。你還可以用常規的方式指定回傳值與設定所需的屬性。

此外，mock 還提供了一個*patch()* 裝飾器，用於 patching 測試範圍🄵對 module（模組）以及 class（類🄵）級🄵的屬性，以及用於建立唯一物件的*sentinel*。有關如何使用*Mock*、*MagicMock* 和*patch()* 的一些範例，請參🄵*快速導引*。

Mock 被設計用於與*unittest* 一起使用，🄵且基於「action（操作）-> assertion（判定）」模式，而不是許多 mocking 框架使用的「record（記🄵）-> replay（重播）」模式。

對於早期版本的 Python，有一個 backport（向後移植的）*unittest.mock* 可以使用，可從 PyPI 下載 mock。

26.6.1 快速導引

Mock 和*MagicMock* 物件在你存取它們時建立所有屬性和 method（方法），🄵儲存它們如何被使用的詳細訊息。你可以配置它們，以指定回傳值或限制可用的屬性，然後對它們的使用方式做出判定：

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect 允許你執行 side effects，包含在 mock 被呼叫時引發例外：

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock 有許多其他方法可以讓你配置與控制它的行🄵。例如，*spec* 引數可以配置 mock ，讓其從另一個物件取得規格。嘗試讀取 mock 中不存在於規格中的屬性或方法將會失敗，🄵出現*AttributeError*。

patch() 裝飾器 / 情境管理器可以在測試中簡單的 mock 模組中的類🄵或物件。被指定的物件在測試期間會被替🄵🄵 mock（或其他物件），🄵在測試結束時恢復：

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

備🄵

當你巢狀使用 patch 裝飾器時，mock 傳遞到被裝飾函式的順序會跟其被應用的順序相同（一般 *Python* 應用裝飾器的順序）。這意味著由下而上，因此在上面的範例中，`module.ClassName1` 的 mock 會先被傳入。

使用*patch()* 時，需注意的是你得在被查找物件的命名空間中（in the namespace where they are looked up）patch 物件。這通常很直接，但若需要快速導引，請參🄵*該 patch 何處*。

裝飾器*patch()* 也可以在 with 陳述式中被用來作🄵情境管理器：

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

也有*patch.dict()*，用於在測試範圍中設定 dictionary（字典）🄵的值，🄵在測試結束時將其恢復🄵原始狀態：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock 支援對 Python 的*魔術方法*的 mocking。最簡單使用魔術方法的方式是使用*MagicMock* 類🄵。它允許你執行以下操作：

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock 允許你將函式（或其他 Mock 實例）分配給魔術方法，🄵且它們將被適當地呼叫。*MagicMock* 類🄵是一個 Mock 的變體，它🄵你預先建好了所有魔術方法（好吧，所有有用的方法）。

以下是在一般 Mock 類🄵中使用魔術方法的範例：

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

🄵了確保測試中的 mock 物件與它們要替🄵的物件具有相同的 api，你可以使用*自動規格*。自動規格（auto-speccing）可以通過 patch 的 *autospec* 引數或*create_autospec()* 函式來完成。自動規格建立的

mock 物件與它們要替🄵的物件具有相同的屬性和方法，🄵且任何函式和方法（包括建構函式）都具有與真實物件相同的呼叫簽名（call signature）。

這可以確保如果使用方法錯誤，你的 mock 會跟實際程式碼以相同的方式失敗：

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

create_autospec() 也可以用在類🄵上，它🄵🄵了 __init__ 方法的簽名，它也可以用在可呼叫物件上，其🄵🄵了 __call__ 方法的簽名。

26.6.2 Mock 類🄵

Mock 是一個彈性的的 mock 物件，旨在代替程式碼中 stubs 和 test doubles （測試替身）的使用。Mock 是可呼叫的，🄵在你存取它們時將屬性建立🄵新的 mock[1]。存取相同的屬性將永遠回傳相同的 mock。Mock 記🄵了你如何使用它們，允許你判定你的程式碼對 mock 的行🄵。

MagicMock 是*Mock* 的子類🄵，其中所有魔術方法均已預先建立🄵可供使用。也有不可呼叫的變體，在你 mock 無法呼叫的物件時很有用：*NonCallableMock* 和*NonCallableMagicMock*

patch() 裝飾器可以輕松地用*Mock* 物件臨時替🄵特定模組中的類🄵。預設情🄵下，*patch()* 會🄵你建立一個*MagicMock*。你可以使用*patch()* 的 *new_callable* 引數指定*Mock* 的替代類🄵。

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

建立一個新的*Mock* 物件。*Mock* 接受數個可選的引數來指定 Mock 物件的行🄵：

- *spec*：這可以是字串的 list（串列），也可以是充當 mock 物件規格的現有物件（類🄵或實例）。如果傳入一個物件，則通過對該物件呼叫 dir 來形成字串的串列（不包括不支援的魔術屬性和方法）。存取不在此串列中的任何屬性都會引發*AttributeError*。

  如果 *spec* 是一個物件（而不是一個字串的串列），那🄵 __class__ 會回傳 spec 物件的類🄵。這允許 mocks 通過*isinstance()* 測試。

- *spec_set*：*spec* 的一個更嚴格的變體。如果使用 *spec_set*，在 mock 上嘗試 *set* 或取得不在傳遞給 *spec_set* 的物件上的屬性將會引發*AttributeError*。

- *side_effect*：每當呼叫 Mock 時要呼叫的函式，參見*side_effect* 屬性，用於引發例外或動態變更回傳值。該函式使用與 mock 相同的引數呼叫，且除非它回傳*DEFAULT*，否則此函式的回傳值將用作回傳值。

  side_effect 也可以是一個例外的類🄵或實例。在這種情🄵下，當呼叫 mock 時，該例外將被引發。

  如果 *side_effect* 是一個可🄵代物件，那🄵對 mock 的每次呼叫將回傳可🄵代物件中的下一個值。

  side_effect 可以通過將其設置🄵 None 來清除。

- *return_value*：當呼叫 mock 時回傳的值。預設情🄵下，這是一個新的 Mock（在首次存取時建立）。參見*return_value* 屬性。

[1] 唯一的例外是魔術方法和屬性（具有前後雙底🄵）。Mock 不會建立這些，而是會引發*AttributeError*。這是因🄵直譯器通常會隱式地要求這些方法，在期望得到一個魔術方法🄵獲得一個新的 Mock 物件時，會讓直譯器非常困惑。如果你需要魔術方法的支援，請參🄵*魔術方法*。

- *unsafe*：預設情🄵下，存取任何以 *assert*、*assret*、*asert*、*aseert* 或 *assrt* 開頭的屬性將引發*AttributeError*。如果傳遞 unsafe=True，將會允許存取這些屬性。

  在 3.5 版被加入.

- *wraps*：被 mock 物件包裝的項目。如果 *wraps* 不是 None，那🄵呼叫 Mock 將通過被包裝的物件（回傳真實結果）。存取 mock 的屬性將會回傳一個 Mock 物件，該物件包裝了被包裝物件的對應屬性（因此嘗試存取不存在的屬性將引發*AttributeError*）。

  如果 mock 已經明確設定了 *return_value*，那🄵呼叫🄵不會被傳遞到被包裝的物件，而是回傳已設定好的 *return_value*。

- *name*：如果 mock 有一個名稱，那🄵它會被用於 mock 的 repr。這對於除錯很有用。此名稱將被傳播到子 mocks。

Mocks 還可以使用任意的關鍵字引數進行呼叫。這些關鍵字引數將在建立 mock 之後用於設定 mock 的屬性。欲知更多，請參見*configure_mock()* 方法。

assert_called()

確認 mock 至少被呼叫一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

在 3.6 版被加入.

assert_called_once()

確認 mock 只被呼叫一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
Calls: [call(), call()].
```

在 3.6 版被加入.

assert_called_with(**args*, ***kwargs*)

這個方法是一個便利的方式，用來斷言最後一次呼叫是以特定方式進行的：

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)

確認 mock 只被呼叫一次，且該次呼叫使用了指定的引數。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
```

(繼續下一頁)

(繼續上一頁)

```
AssertionError: Expected 'mock' to be called once. Called 2 times.
Calls: [call('foo', bar='baz'), call('other', bar='values')].
```

assert_any_call(**args*, ***kwargs*)

斷言 mock 已經被使用指定的引數呼叫。

這個斷言在 mock 曾經被呼叫過時通過，不同於*assert_called_with()*和*assert_called_once_with()*，他們針對的是最近的一次的呼叫，而且對於*assert_called_once_with()*，最近一次的呼叫還必須也是唯一一次的呼叫。

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

斷言 mock 已經使用指定的呼叫方式來呼叫。此斷言會檢查*mock_calls*串列中的呼叫。

如果 *any_order* 🄵 false，那🄵這些呼叫必須按照順序。在指定的呼叫之前或之後可以有額外的呼叫。

如果 *any_order* 🄵 true，那🄵這些呼叫可以以任何順序出現，但它們必須全部出現在*mock_calls*中。

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

斷言 mock 從未被呼叫。

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
Calls: [call()].
```

在 3.5 版被加入.

reset_mock(***, *return_value=False*, *side_effect=False*)

reset_mock 方法重置 mock 物件上的所有呼叫屬性:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

This can be useful where you want to make a series of assertions that reuse the same object.

return_value parameter when set to `True` resets *return_value*:

```
>>> mock = Mock(return_value=5)
>>> mock('hello')
5
>>> mock.reset_mock(return_value=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

side_effect parameter when set to `True` resets *`side_effect`*:

```
>>> mock = Mock(side_effect=ValueError)
>>> mock('hello')
Traceback (most recent call last):
  ...
ValueError
>>> mock.reset_mock(side_effect=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

Note that *`reset_mock()`* *doesn't* clear the *`return_value`*, *`side_effect`* or any child attributes you have set using normal assignment by default.

Child mocks are reset as well.

在 3.6 版的變更: reset_mock 函式新增了兩個僅限關鍵字引數 (keyword-only arguments)。

mock_add_spec(*spec*, *spec_set=False*)

向 mock 增加一個規格 (spec)。*spec* 可以是一個物件或一個字串串列 (list of strings)。只有在 *spec* 上的屬性才能作🄵 mock 的屬性被取得。

如果 *spec_set* 🄵 true，那🄵只能設定在規格中的屬性。

attach_mock(*mock*, *attribute*)

將一個 mock 作🄵這個 Mock 的屬性附加，取代它的名稱和上代 (parent)。對附加的 mock 的呼叫將被記🄵在這個 Mock 的*`method_calls`* 和*`mock_calls`* 屬性中。

configure_mock(***kwargs*)

透過關鍵字引數在 mock 上設定屬性。

可以在使用 method（方法）呼叫時，使用標準點記法 (dot notation) 🄵將字典拆開，🄵 child mock 設定屬性、回傳值和 side effects：

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

同樣的事情可以在 mock 的建構函式呼叫中實現：

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() 的存在是🄵了在 mock 被建立後更容易進行組態設定。

__dir__()

Mock 物件限制了 `dir(some_mock)` 僅顯示有用的結果。對於具有 *spec* 的 mock，這包含所有被允許的 mock 屬性。

請參🄵*FILTER_DIR* 以了解這種過濾行🄵的作用，以及如何關閉它。

_get_child_mock(***kw*)

建立🄵了得到屬性和回傳值的 child mock。預設情🄵下，child mock 將與其上代是相同的型🄵。Mock 的子類🄵可能會想要置🄵此行🄵，以自定義 child mock 的建立方式。

對於不可呼叫的 mock，將使用可呼叫的變體，而不是任何的自定義子類🄵。

called

一個 boolean（布林），表述 mock 物件是否已經被呼叫：

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

一個整數，告訴你 mock 物件被呼叫的次數：

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

設定此值以配置呼叫 mock 時回傳的值：

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

預設的回傳值是一個 mock 物件，你也可以按照正常的方式配置它：

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value 也可以在建構函式中設定：

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

這可以是一個在呼叫 mock 時要呼叫的函式、一個可🄵代物件，或者要引發的例外（類🄵或實例）。

如果你傳遞一個函式，它將被呼叫，其引數與 mock 相同，且除非該函式回傳*DEFAULT* 單例 (singleton)，否則對 mock 的呼叫將回傳函式回傳的任何值。如果函式回傳*DEFAULT*，那🄵 mock 將回傳其正常的回傳值（從*return_value* 得到）。

如果你傳遞一個可🄵代物件，它將被用於檢索一個🄵代器，該🄵代器必須在每次呼叫時🄵出 (yield) 一個值。這個值可以是要引發的例外實例，或者是對 mock 呼叫時要回傳的值（處理*DEFAULT* 的方式與函式的狀🄵相同）。

以下是一個引發例外的 mock 的範例（用於測試 API 的例外處理）：

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

使用*side_effect* 回傳一連串值的範例：

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

使用可被呼叫物件的範例：

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect 可以在建構函式中設定。以下是一個範例，它將 mock 被呼叫時給的值加一🄵回傳：

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

將*side_effect* 設定🄵 `None` 可以清除它：

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

這會是 `None`（如果 mock 尚未被呼叫），或是 mock 上次被呼叫時使用的引數。這將以元組的形式呈現：第一個成員 (member)，其可以通過 `args` 屬性存取，是 mock 被呼叫時傳遞的所有位置引數（或一個空元組）。第二個成員，其可以通過 `kwargs` 屬性存取，是所有關鍵字引數（或一個空字典）。

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args，以及串列*call_args_list*、*method_calls* 和*mock_calls* 的成員都是*call* 物件。這些都是元組，因此可以解包以取得各個引數🄴進行更🄵雜的斷言。參見*calls as tuples*。

在 3.8 版的變更: 新增 `args` 與 `kwargs` 特性。

call_args_list

這是按順序列出所有呼叫 mock 物件的串列（因此串列的長度表示它被呼叫的次數）。在任何呼叫發生之前，它會是一個空的串列。*call* 物件可用於方便地建構呼叫的串列，以便與*call_args_list* 進行比較。

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

call_args_list 的成員都是*call* 物件。這些物件可以被拆包🄴元組，以取得各個引數。參見*calls as tuples*。

method_calls

除了追🄴對自身的呼叫之外，mock 還會追🄴對方法和屬性的呼叫，以及它們（這些方法和屬性）的方法和屬性：

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

method_calls 的成員都是*call* 物件。這些物件可以拆包🄴元組，以取得各個引數。參見*calls as tuples*。

mock_calls

mock_calls 記🄵了 所有對 mock 物件的呼叫，包含其方法、魔術方法以及回傳值 mock。

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

method_calls 的成員都是*call* 物件。這些物件可以拆包🄵元組，以取得各個引數。參見*calls as tuples*。

備🄵

mock_calls 記🄵的方式意味著在進行巢狀呼叫時，上代 (ancestor) 呼叫的參數不會被記🄵，因此在比較時它們將始終相等：

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

通常，物件的 __class__ 屬性會回傳它的型🄵。但對於擁有 spec 的 mock 物件，__class__ 會回傳 spec 的類🄵。這允許 mock 物件通過對它們所替代或🄵裝的物件進行的*isinstance()* 測試：

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ 可以被指定，這允許 mock 通過*isinstance()* 檢查，而不需要🄵制使用 spec：

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

Mock 的一個不可呼叫版本。建構函式參數的意義與*Mock* 相同，其例外🄵 *return_value* 和 *side_effect* 在不可呼叫的 mock 上無意義。

使用類🄵或實例作🄵 spec 或 spec_set 的 mock 物件能🄵通過*isinstance()* 測試：

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
```

(繼續下一頁)

(繼續上一頁)

```
>>> isinstance(mock, SomeClass)
True
```

Mock 類🄵支援 mock 魔術方法。細節請參考*魔術方法*。

Mock 類🄵和*patch()* 裝飾器於組態時接受任意的關鍵字引數。對於*patch()* 裝飾器，這些關鍵字會傳遞給正在建立 mock 的建構函式。這些關鍵字引數用於配置 mock 的屬性：

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

Child mock 的回傳值和 side effect 可以使用使用點記法進行設置。由於你無法直接在呼叫中使用帶有點 (.) 的名稱，因此你必須建立一個字典🄵使用 `**` 解包：

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

在匹配對 mock 的呼叫時，使用 *spec*（或 *spec_set*）建立的可呼叫 mock 將會🄵省 (introspect) 規格物件的簽名 (signature)。因此，它可以匹配實際呼叫的引數，無論它們是按位置傳遞還是按名稱傳遞：

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

這適用於*assert_called_with()*、*assert_called_once_with()*、*assert_has_calls()* 和*assert_any_call()*。在使用*Autospeccing*（自動規格）時，它還適用於 mock 物件的方法呼叫。

在 3.4 版的變更: 對於已經設置了規格（spec）和自動規格（autospec）的 mock 物件，新增簽名🄵省功能。

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

一個理應在類🄵上當成*property* 或其他*descriptor* 的 mock。*PropertyMock* 提供了 `__get__()` 和 `__set__()` 方法，因此你可以在它被提取時指定回傳值。

從物件中提取*PropertyMock* 實例會不帶任何引數呼叫 mock。設定它則會用設定的值來呼叫 mock：

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
```

(繼續下一頁)

(繼續上一頁)

```
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

由於 mock 屬性的儲存方式，你無法直接將*PropertyMock* 附加到 mock 物件。但是你可以將其附加到 mock 型🄵的物件：

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

警示

If an *AttributeError* is raised by *PropertyMock*, it will be interpreted as a missing descriptor and `__getattr__()` will be called on the parent mock:

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

詳情請見 `__getattr__()`。

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

MagicMock 的非同步版本。*AsyncMock* 物件的表現將被視🄵非同步函式，🄵且呼叫的結果是一個可等待物件。

```
>>> mock = AsyncMock()
>>> inspect.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

`mock()` 的結果是一個非同步函式，在它被等待後將具有 `side_effect` 或 `return_value` 的結果：

- 如果 `side_effect` 是一個函式，非同步函式將回傳該函式的結果，
- 如果 `side_effect` 是一個例外，則非同步函式將引發該例外，
- 如果 `side_effect` 是一個可🄵代物件，非同步函式將回傳可🄵代物件的下一個值，但如果結果序列耗盡，將立即引發 `StopAsyncIteration`，
- 如果 `side_effect` 🄵有被定義，非同步函式將回傳由 `return_value` 定義的值，因此在預設情🄵下，非同步函式回傳一個新的*AsyncMock* 物件。

將*Mock* 或*MagicMock* 的 *spec* 設定🄵非同步函式將導致在呼叫後回傳一個協程物件。

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
```

(繼續下一頁)

(繼續上一頁)

```
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

將*Mock*、*MagicMock* 或*AsyncMock* 的 *spec* 設定🄵具有同步和非同步函式的類🄵，會自動檢測同步函式🄵將其設定🄵*MagicMock*（如果上代 mock 🄵*AsyncMock* 或*MagicMock*）或*Mock*（如果上代 mock 🄵*Mock*）。所有非同步函式將被設定🄵*AsyncMock*。

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

在 3.8 版被加入.

assert_awaited()

斷言 mock 至少被等待過一次。請注意這與物件是否被呼叫是分開的，`await` 關鍵字必須被使用：

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

斷言 mock 正好被等待了一次。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

斷言最後一次等待使用了指定的引數。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

斷言 mock 只被等待了一次🄵使用了指定的引數。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

斷言 mock 曾經被使用指定的引數等待過。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

斷言 mock 已被使用指定的呼叫進行等待。*await_args_list* 串列將被檢查以確認等待的🄵容。

如果 *any_order* 🄵 false，則等待必須按照順序。指定的等待之前或之後可以有額外的呼叫。

如果 *any_order* 🄵 true，則等待可以以任何順序出現，但它們必須全部出現在*await_args_list* 中。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
```

(繼續下一頁)

(繼續上一頁)

```
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

斷言 mock 從未被等待。

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

參見*Mock.reset_mock()*。同時將*await_count* 設定🄵 0，*await_args* 設定🄵 None，🄵清除*await_args_list*。

await_count

一個整數，用來記🄵 mock 物件已被等待的次數。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

這可能是 None（如果 mock 尚未被等待），或者是上次等待 mock 時使用的引數。與*Mock.call_args* 的功能相同。

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

這是一個按照順序記🄵 mock 物件所有等待的串列（因此串列的長度表示該物件已被等待的次數）。在進行任何等待之前，此串列🄵空。

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
```

(繼續下一頁)

(繼續上一頁)

```
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.**ThreadingMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, *, *timeout=UNSET*, ***kwargs*)

A version of *MagicMock* for multithreading tests. The *ThreadingMock* object provides extra methods to wait for a call to be invoked, rather than assert on it immediately.

The default timeout is specified by the `timeout` argument, or if unset by the *ThreadingMock.DEFAULT_TIMEOUT* attribute, which defaults to blocking (`None`).

You can configure the global default timeout by setting *ThreadingMock.DEFAULT_TIMEOUT*.

wait_until_called(*, *timeout=UNSET*)

等待直到 mock 被呼叫。

If a timeout was passed at the creation of the mock or if a timeout argument is passed to this function, the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(*args*, ***kwargs*)

等到直到 mock 被以特定引數呼叫。

If a timeout was passed at the creation of the mock the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1", "arg2",), kwargs={"arg":
↪"thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2", arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

Global default timeout in seconds to create instances of *ThreadingMock*.

在 3.13 版被加入.

呼叫

Mock 物件可被呼叫。呼叫將回傳設定🄵*return_value* 屬性的值。預設的回傳值是一個新的 Mock 物件；它會在第一次存取回傳值時（無論是顯式存取還是透過呼叫 Mock）被建立，但是這個回傳值會被儲存，之後每次都回傳同一個值。

對物件的呼叫會被記🄵在如*call_args* 和*call_args_list* 等屬性中。

如果*side_effect* 被設定，那🄵在呼叫被記🄵後它才會被呼叫，所以如果 `side_effect` 引發例外，呼叫仍然會被記🄵。

呼叫 mock 時引發例外的最簡單方式是將*side_effect* 設定🄵例外類🄵或實例：

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
```

(繼續下一頁)

(繼續上一頁)

```
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

如果*side_effect* 是一個函式，則該函式回傳的東西就是對 mock 的呼叫所回傳的值。side_effect 函式會使用與 mock 相同的引數被呼叫。這讓你可以根據輸入動態地變更呼叫的回傳值：

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

如果你希望 mock 仍然回傳預設的回傳值（一個新的 mock），或者是任何已設定的回傳值，有兩種方法可以實現。從*side_effect* 🄵部回傳*return_value*，或回傳*DEFAULT*：

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

要🄵除*side_effect*，🄵恢復預設行🄵，將 side_effect 設🄵 None：

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

side_effect 也可以是任何可🄵代的物件。對 mock 的重🄵呼叫將從可🄵代物件中回傳值（直到🄵代物件耗盡🄵引發*StopIteration* 🄵止）：

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
```

(繼續下一頁)

(繼續上一頁)

```
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

如果可🄵代物件中的任何成員是例外，則它們將被引發而不是被回傳：

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

🄵除屬性

Mock 物件會在需要時建立屬性。這使得它們可以假裝成任何種類的物件。

你可能希望一個 mock 物件在*hasattr()* 呼叫時回傳 `False`，或者在屬性被提取時引發*AttributeError*。你可以通過將物件提供🄵 mock 的 `spec` 來實現這一點，但這🄵不總是那🄵好用。

你可以通過🄵除屬性來「阻擋」它們。一旦🄵除，再次存取該屬性將會引發*AttributeError*。

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock 名稱與名稱屬性

由於”name” 是傳遞給*Mock* 建構函式的引數，如果你想讓你的 mock 物件擁有”name” 屬性，你不能在建立時直接傳遞它。有兩種替代方法。其中一個選擇是使用*configure_mock()*：

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

更簡單的方法是在 mock 建立後直接設定”name” 屬性：

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

如同屬性一般附加 mock

當你將一個 mock 附加🄵另一個 mock 的屬性（或作🄵回傳值），它將成🄵該 mock 的「子代 (child)」。對子代的呼叫將被記🄵在上代的*method_calls* 和*mock_calls* 屬性中。這對於配置子代🄵將它們附加到上代，或將 mock 附加到記🄵所有對子代的呼叫的上代🄵允許你對 mock 間的呼叫順序進行斷言非常有用：

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

如果 mock 有 name 引數，則上述規則會有例外。這使你可以防止「親屬關🄵 (parenting)」的建立，假設因🄵某些原因你不希望這種狀🄵發生。

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

由*patch()* 🄵你建立的 mock 會自動被賦予名稱。若要將具有名稱的 mock 附加到上代，你可以使用*attach_mock()* 方法：

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.6.3 Patchers

patch 裝飾器僅用於在裝飾的函式範圍🄵對物件進行 patch。它們會自動🄵你處理 patch 的中止，即使有🄵常被引發也是如此。所有這些函式也可以在 with 陳述式中使用，或者作🄵類🄵裝飾器使用。

patch

> **備🄵**
>
> 關鍵是要在正確的命名空間進行 patch。請參🄵*where to patch* 一節。

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() 充當函式裝飾器、類🄵裝飾器或情境管理器。在函式或 with 陳述式的🄵部，目標會被 patch 成一個新的物件。當函式或 with 陳述式結束時，patch 就會被解除。

如果 *new* 被省略，則如果被 patch 的物件是非同步函式，目標會被替🄵🄵*AsyncMock*，反之則替🄵🄵*MagicMock*。如果*patch()* 做🄵裝飾器使用且省略了 *new*，則所建立的 mock 會作🄵額外的引數傳遞給被裝飾的函式。如果*patch()* 作🄵情境管理器使用，則所建立的 mock 將由情境管理器回傳。

target 應該是以 `'package.module.ClassName'` 形式出現的字串。*target* 會被引入🄵用 *new* 物件替🄵指定的物件，因此 *target* 必須可從你呼叫*patch()* 的環境中引入。target 在執行被裝飾的函式時被引入，而不是在裝飾器作用時 (decoration time)。

spec 和 *spec_set* 關鍵字引數會傳遞給*MagicMock*，如果 patch 正在🄵你建立一個。

此外，你還可以傳遞 `spec=True` 或 `spec_set=True`，這將導致 patch 將被 mock 的物件作🄵 spec/spec_set 物件傳遞。

new_callable 允許你指定一個不同的類🄵或可呼叫的物件，用於被呼叫🄵建立 *new* 物件。預設情🄵下，對於非同步函式使用*AsyncMock*，而對於其他情🄵則使用*MagicMock*。

spec 的一種更🄵大的形式是 *autospec*。如果你設定 `autospec=True`，則該 mock 將使用被替🄵物件的規格來建立。該 mock 的所有屬性也將具有被替🄵物件的對應屬性的規格。被 mock 的方法和函式將檢查其引數，如果呼叫時引數與規格不符（被使用錯誤的簽名 (signature) 呼叫），將引發*TypeError*。對於替🄵類🄵的 mock，它們的回傳值（即'instance'）將具有與類🄵相同的規格。請參🄵*create_autospec()* 函式和*Autospeccing（自動規格）*。

你可以用 `autospec=some_object` 替代 `autospec=True`，以使用任意物件作🄵規格，而不是被替🄵的物件。

預設情🄵下，*patch()* 將無法取代不存在的屬性。如果你傳入 `create=True`，且屬性不存在，則當被 patch 的函式被呼叫時，patch 將🄵你建立該屬性，🄵在被 patch 的函式結束後再次🄵除它。這對於撰寫針對你的生🄵程式碼在執行環境建立的屬性的測試時非常有用。此功能預設🄵關閉，因🄵這可能會相當危險。開🄵這個功能後，你可以對於實際上不存在的 API 撰寫會通過的測試！

> **備🄵**
>
> 在 3.5 版的變更: 如果你正在 patch 模組中的🄵建函式，那🄵你不需要傳遞 `create=True`，它預設會被加入。

patch 可以做🄵*TestCase* 類🄵的裝飾器使用。它透過裝飾類🄵中的每個測試方法來運作。當你的測試方法共享一組常見的 patch 時，這會🄵少繁冗的代碼。*patch()* 通過搜尋以 `patch.TEST_PREFIX` 開頭的方法名來尋找測試。預設情🄵下會是 `'test'`，這與*unittest* 尋找測試的方式相匹配。你可以通過設定 `patch.TEST_PREFIX` 來指定🄵的前綴。

透過 with 陳述式，Patch 可以做🄵情境管理器使用。patch 適用於 with 陳述式之後的縮排區塊。如果你使用"as"，則被 patch 的物件將被綁定到"as" 後面的名稱；如果*patch()* 正在🄵你建立一個 mock 物件，這會非常有用。

patch() 接受任意的關鍵字引數。如果被 patch 的物件是非同步的，這些將會被傳遞給*AsyncMock*，如果是同步的則會傳遞給*MagicMock*，或如果指定了 *new_callable*，則傳遞給它。

`patch.dict(...)`、`patch.multiple(...)` 和 `patch.object(...)` 可用於其余的使用情境。

patch() 作🄵函式裝飾器，🄵你建立 mock 🄵將其傳遞給被裝飾的函式：

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patch 一個類🄵會以*MagicMock* 實例取代該類🄵。如果該類🄵在被測試的程式碼中實例化，那🄵它將是會被使用的 mock 的*return_value*。

如果該類🄵被實例化多次，你可以使用*side_effect* 來每次回傳一個新的 mock。或者你可以將 *return_value* 設定成你想要的任何值。

若要配置被 patch 的類🄵的實例方法的回傳值，你必須在*return_value* 上進行配置。例如：

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

如果你使用 *spec* 或 *spec_set* 且*patch()* 正在取代一個類🄵，那🄵被建立的 mock 的回傳值將具有相同的規格。：

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

當你想要🄵被建立的 mock 使用一個替代的類🄵取代預設的*MagicMock* 時，*new_callable* 引數非常有用。例如，如果你想要一個*NonCallableMock* 被使用：

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

另一個用法是用一個*io.StringIO* 實例替🄵一個物件：

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

當*patch()* 🄵你建立 mock 時，通常你需要做的第一件事就是配置 mock。其中一些配置可以在對 patch 的呼叫中完成。你傳遞到呼叫中的任何關鍵字都將用於在被建立的 mock 上設定屬性：

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

除了被建立的 mock 上的屬性外，還可以配置 child mock 的*return_value* 和*side_effect*。它們在語法上不能直接作🄵關鍵字引數傳入，但是以它們作🄵鍵的字典仍然可以使用 ** 擴充🄵一個*patch()* 呼叫：

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

預設情🄵下，嘗試 patch 模組中不存在的函式（或類🄵中的方法或屬性）將會失敗，🄵引發*AttributeError*：

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing_attribute
↪'
```

但是在對*patch()* 的呼叫中增加 `create=True` 將使前面的範例按照預期運作：

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

在 3.8 版的變更: 如果目標是一個非同步函式，*patch()* 現在會回傳一個*AsyncMock*。

patch.object

`patch.`**`object`**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

使用一個 mock 物件 patch 一個物件（目標）上的命名成員（屬性）。

patch.object() 可以做🄵裝飾器、類🄵裝飾器或情境管理器使用。引數 *new*、*spec*、*create*、*spec_set*、*autospec* 和 *new_callable* 與在*patch()* 中的引數具有相同的意義。與*patch()* 一樣，*patch.object()* 接受任意關鍵字引數來配置它所建立的 mock 物件。

當作🄵類🄵裝飾器使用時，*patch.object()* 會遵循 `patch.TEST_PREFIX` 來選擇要包裝的方法。

你可以使用三個引數或兩個引數來呼叫*patch.object()*。三個引數的形式接受要被 patch 的物件、屬性名稱和要替🄵掉屬性的物件。

當使用兩個引數的形式呼叫時，你會省略要替🄵的物件，一個 mock 會🄵你建立🄵將其作🄵額外的引數傳遞給被裝飾的函式：

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec、*create* 和*patch.object()* 的其他引數與在*patch()* 中的引數具有相同的意義。

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch 字典或類字典的物件，🄵在測試後將字典回復到其原本的狀態，其中回復的字典是測試前字典的🄵本。

in_dict 可以是一個字典或一個類對映的容器。如果它是一個對映，那🄵它至少必須支援取得、設定、🄵除項目以及對鍵的🄵代。

in_dict 也可以是指定字典名稱的字串，然後透過 import 來取得該字典。

values 可以是要設定的值的字典。*values* 也可以是 `(key, value)` 對 (pairs) 的可🄵代物件。

如果 *clear* 🄵 true，則在設定新值之前字典將被清除。

也可以使用任意關鍵字引數呼叫*patch.dict()* 以在字典中設定值。

在 3.8 版的變更: *patch.dict()* 現在在做🄵情境管理器使用時回傳被 patch 的字典。

patch.dict() 可以做🄵情境管理器、裝飾器或類🄵裝飾器使用：

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

當作🄵類🄵裝飾器使用時，*patch.dict()* 會遵循 patch.TEST_PREFIX（預設🄵 'test'）來選擇要包裝的方法：

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

如果你想在測試中使用不同的前綴，你可以透過設定 patch.TEST_PREFIX 來告知 patcher 使用不同的前綴。請參🄵*TEST_PREFIX* 以得知如何修改前綴的更多🄵容。

patch.dict() 可用於在字典中新增成員，或單純地讓測試更改字典，🄵確保在測試結束時將字典回復原狀。

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

可以在*patch.dict()* 呼叫中使用關鍵字來設定字典中的值：

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() 可以與實際上不是字典的類字典物件一起使用。最低限度它們必須支援項目的取得、設定、🄵除以及🄵代或隸屬資格檢測。這對應到魔術方法中的 `__getitem__()`、`__setitem__()`、`__delitem__()` 以及*__iter__()* 或 `__contains__()`。

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

在一次呼叫中執行多個 patch。它接受被 patch 的物件（作🄵物件或透過 import 取得物件的字串）和 patch 的關鍵字引數：

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

如果你想要*patch.multiple()* 🄵你建立 mock，請使用*DEFAULT* 作🄵值。在這種情🄵下，被建立的 mock 會透過關鍵字傳遞到被裝飾的函式中，🄵且當*patch.multiple()* 作🄵情境管理器時會回傳字典。

patch.multiple() 可以做🄵裝飾器、類🄵裝飾器或情境管理器使用。引數 *spec*、*spec_set*、*create*、*autospec* 和 *new_callable* 與在*patch()* 中的引數具有相同的意義。這些引數將應用於由*patch.multiple()* 完成的所有 patch。

當作🄵類🄵裝飾器使用時，*patch.multiple()* 遵循 `patch.TEST_PREFIX` 來選擇要包裝的方法。

如果你想要*patch.multiple()* 🄵你建立 mock，那🄵你可以使用*DEFAULT* 作🄵值。如果你使用*patch.multiple()* 作🄵裝飾器，那🄵被建立的 mock 將透過關鍵字傳遞到被裝飾的函式中。：

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
```

(繼續下一頁)

(繼續上一頁)

```
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() 可以與其他 patch 裝飾器巢狀使用，但需要將透過關鍵字傳遞的引數放在*patch()* 建立的任何標准引數之後：

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

如果*patch.multiple()* 作🄴情境管理器使用，則情境管理器回傳的值是一個字典，其中被建立的 mock 會按名稱作🄴其鍵值：

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch 方法：🄵動與停止

所有的 patcher 都有 `start()` 與 `stop()` 方法。這使得在 `setUp` 方法中進行 patch 或在你想要在🄴有巢狀使用裝飾器或 with 陳述式的情🄵下進行多個 patch 時變得更簡單。

要使用它們，請像平常一樣呼叫*patch()*、*patch.object()* 或*patch.dict()*，🄵保留對回傳的 `patcher` 物件的參照。之後你就可以呼叫 `start()` 將 patch 准備就緒，🄵呼叫 `stop()` 來取消 patch。

如果你使用*patch()* 🄵你建立 mock，那🄵它將透過呼叫 `patcher.start` 回傳。：

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

一個典型的用法是在一個*TestCase* 的 `setUp` 方法中執行多個 patch：

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
```

(繼續下一頁)

(繼續上一頁)

```
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

警示

如果你使用這個技巧，你必須確保透過呼叫 `stop` 來"取消" patch。這可能會比你想像的還要🄵雜一點，因🄵如果有例外在 `setUp` 中被引發，則 `tearDown` 就不會被呼叫。*unittest.TestCase.addCleanup()* 會讓這稍微簡單一點：

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

作🄵額外的好處，你不再需要保留對 `patcher` 物件的參照。

也可以使用*patch.stopall()* 來停止所有已🄵動的 patch。

patch.**stopall**()

停止所有運作的 patch。只停止以 `start` 🄵動的 patch。

patch 🄵建函式

你可以 patch 模組🄵的任何🄵建函式。以下範例 patch 🄵建函式*ord()*：

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

所有 patcher 都可以作🄵類🄵裝飾器使用。以這種方式使用時，它們包裝了類🄵上的每個測試方法。Patcher 將 `'test'` 開頭的方法認定🄵測試方法。這與*unittest.TestLoader* 預設尋找測試方法的方式相同。

你可能會想🄵你的測試使用不同的前綴。你可以透過設定 `patch.TEST_PREFIX` 來告知 patcher 使用不同的前綴：

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
```

(繼續下一頁)

(繼續上一頁)

```
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

巢狀使用 Patch 裝飾器

如果你想執行多個 patch，那🄵你可以簡單地堆🄵裝飾器。

你可以使用這個模式來堆🄵多個 patch 裝飾器：

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

請注意，裝飾器是從底部向上應用的。這是 Python 應用裝飾器的標準方式。被建立的 mock 傳遞到測試函式中的順序與此順序相同。

該 patch 何處

patch() 的工作原理是（暫時）將 *name* 指向的物件變更🄵另一個物件。可以有許多 name 指向任何單一物件，因此🄵了使 patch 起作用，你必須確保你 patch 了被測試系統使用的 name。

基本原則是在物件被查找的位置進行 patch，該位置不一定與其被定義的位置相同。幾個範例將有助於闡明這一點。

想像一下，我們想要測試一個專案，其結構如下：

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

現在我們想要測試 `some_function`，但我們想使用*patch()* mock `SomeClass`。問題是，當我們 import 模組 b 時（我們必須這樣做），它會從模組 a import `SomeClass`。如果我們使用*patch()* 來 mock `a.SomeClass`，那🄵它對我們的測試就不會有任何影響；模組 b 已經有了一個真實的 `SomeClass` 的參照，看起來我們的 patch 🄵🄵有任何效果。

關鍵是在使用（或查找它）的地方 patch `SomeClass`。在這個情🄵下，`some_function` 實際上會在我們 import 它的模組 b 中查找 `SomeClass`。這🄵的 patch 應該長得像這樣：

```
@patch('b.SomeClass')
```

然而，考慮另一種情🄵，其中模組 b 🄵不是使用 `from a import SomeClass`，而是 `import a`，然後 `some_function` 使用 `a.SomeClass`。這兩種 import 形式都很常見。在這種情🄵下，我們想要 patch 的類🄵正在其模組中被查找，因此我們必須 patch `a.SomeClass`：

```
@patch('a.SomeClass')
```

Patch 描述器與代理物件 (Proxy Objects)

patch 和*patch.object* 都正確地 patch 和還原描述器：類🄵方法、🄵態方法以及屬性。你應該在 類🄵而不是實例上 patch 它們。它們還可以使用代理屬性存取的一些物件，例如 django 設定物件。

26.6.4 MagicMock 以及魔術方法支援

Mock 魔術方法

Mock 支援 mock Python 協定方法，其也被稱作"*魔術方法*"。這允許 mock 物件替🄵容器或實作 Python 協定的其他物件。

由於魔術方法被查找的方式與一般的方法[2] 不同，因此專門實作了此支援方式。這代表著僅有特定的魔術方法被此方式支援。現在已支援清單中已經幾乎包含了所有魔術方法。如果你需要 mock 任何魔術方法而其尚未被支援，請讓我們知道。

你可以透過將你感興趣的方法設定🄵函式或 mock 實例來 mock 魔術方法。如果你使用函式，那🄵它必須將 `self` 作🄵第一個引數[3]。

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

一個用法是在 `with` 陳述式中 mock 作🄵情境管理器使用的物件：

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

對魔術方法的呼叫🄵不會出現在*method_calls* 中，它們會被記🄵在*mock_calls* 🄵。

> **備🄵**
>
> 如果你使用 *spec* 關鍵字引數來建立一個 mock，則嘗試設定規格中未包含的魔術方法將引發一個*AttributeError*。

已支援的魔術方法的完整列表是：

[2] 魔術方法應該在類🄵而不是實例上被查找。不同版本的 Python 對於這條規則的適用🄵不一致。支援的協定方法應適用於所有支援的 Python 版本。

[3] 該函式基本上與類🄵🄵🄵，但每個 `Mock` 實例都與其他實例保持隔離。

- __hash__、__sizeof__、__repr__ 和 __str__
- __dir__、__format__ 和 __subclasses__
- __round__、__floor__、__trunc__ 和 __ceil__
- 比較方法：__lt__、__gt__、__le__、__ge__、__eq__ 和 __ne__
- 容器方法：__getitem__、__setitem__、__delitem__、__contains__、__len__、__iter__、__reversed__ 和 __missing__
- 情境管理器：__enter__、__exit__、__aenter__ 和 __aexit__
- 一元數值方法：__neg__、__pos__ 和 __invert__
- 數值方法（包括右側 (right hand) 和原地 (in-place) 變體）：__add__、__sub__、__mul__、__matmul__、__truediv__、__floordiv__、__mod__、__divmod__、__lshift__、__rshift__、__and__、__xor__、__or__ 和 __pow__
- 數值轉🄵方法：__complex__、__int__、__float__ 和 __index__
- 描述器方法：__get__、__set__ 和 __delete__
- Pickling：__reduce__、__reduce_ex__、__getinitargs__、__getnewargs__、__getstate__ 和 __setstate__
- 檔案系統路徑表示法：__fspath__
- 非同步🄴代方法：__aiter__ 和 __anext__

在 3.8 版的變更: 新增對於*os.PathLike.__fspath__()* 的支援。

在 3.8 版的變更: 新增對於 __aenter__、__aexit__、__aiter__ 和 __anext__ 的支援。

以下方法存在，但「不」被支援，因🄴它們在被 mock 使用時，會無法動態設定，或可能導致問題：

- __getattr__、__setattr__、__init__ 和 __new__
- __prepare__、__instancecheck__、__subclasscheck__、__del__

Magic Mock

MagicMock 有兩個變體：*MagicMock* 和*NonCallableMagicMock*。

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock 是*Mock* 的子類🄵，其預設具有大多數魔術方法的實作。你可以使用 MagicMock，而無需自行配置魔術方法。

建構函式參數的意義與*Mock* 中的參數相同。

如果你使用 *spec* 或 *spec_set* 引數，那🄵只有規格中存在的魔術方法會被建立。

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

MagicMock 的不可呼叫版本。

建構函式參數的意義與*MagicMock* 中的參數相同，但 *return_value* 和 *side_effect* 除外，它們對不可呼叫的 mock 來🄵🄵有任何意義。

魔術方法是使用*MagicMock* 物件設定的，因此你可以配置它們🄴以一般的方法來使用它們：

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

預設情🄴下，許多協定方法都需要回傳特定種類的物件。這些方法預先配置了預設回傳值，因此如果你對回傳值不感興趣，則無需執行任何操作即可使用它們。如果你想更改預設值，你仍然可以手動設定回傳值。

方法及其預設值：

- __lt__: *NotImplemented*
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False
- __len__: 0
- __iter__: iter([])
- __exit__: False
- __aexit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: mock 的預設雜𝔽
- __str__: mock 的預設字串
- __sizeof__: mock 的預設 sizeof

舉例來𝔽：

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

__eq__() 和 __ne__() 這兩個相等的方法是特𝔽的。它們使用*side_effect* 屬性對識𝔽性 (identity) 進行預設的相等比較，除非你變更它們的回傳值以回傳其他𝔽容：

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

MagicMock.__iter__() 的回傳值可以是任何可𝔽代物件，且不需是一個𝔽代器：

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

如果回傳值是一個𝔽代器，那𝔽對其進行一次𝔽代將消耗它，𝔽且後續𝔽代將𝔽生一個空串列：

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

MagicMock 配置了所有支援的魔術方法，除了一些少見和過時的方法。如果你想要，你仍然可以設定這些魔術方法。

MagicMock 中支援但預設未設置的魔術方法包含：

- __subclasses__
- __dir__
- __format__
- __get__、__set__ 和 __delete__
- __reversed__ 和 __missing__
- __reduce__、__reduce_ex__、__getinitargs__、__getnewargs__、__getstate__ 和 __setstate__
- __getformat__

26.6.5 輔助函式

sentinel（哨兵）

unittest.mock.**sentinel**

哨兵物件提供了一種🄵你的測試提供獨特物件的便利方式。

當你使用名稱存取屬性時，屬性會根據需要被建立。存取相同的屬性將始終回傳相同的物件。回傳的物件會具有合適的 repr，讓測試失敗的訊息是可🄵讀的。

在 3.7 版的變更: 哨兵屬性現在當被🄵🄵或序列化時會保留其識🄴性。

在測試時，有時你需要測試特定物件是否作🄴引數被傳遞給另一個方法或回傳。建立命名的哨兵物件來測試這一點是常見的。*sentinel* 提供了一種此類建立和測試物件識🄴性的便利方式。

在這個例子中，我們 monkey patch method 以回傳 sentinel.some_object：

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

DEFAULT 物件是一個預先建立的哨兵（實際上是 sentinel.DEFAULT）。它可以被*side_effect* 函式使用來表示正常的回傳值應該被使用。

call

unittest.mock.**call**(**args*, ***kwargs*)

與*call_args*、*call_args_list*、*mock_calls* 和*method_calls* 相比，*call()* 是一個用於進行更簡單的斷言的輔助物件。*call()* 也可以與*assert_has_calls()* 一起使用。

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

對於表示多個呼叫的 call 物件，*call_list()* 回傳所有中間呼叫以及最終呼叫的串列。

call_list 在對「🄵接呼叫 (chained calls)」進行斷言時特🄵有用。🄵接呼叫是在單行程式碼進行的多次呼叫。這會導致 mock 上的*mock_calls* 中出現多個項目。手動建構呼叫序列會相當單調乏味。

call_list() 可以從同一個🄵接呼叫建構呼叫序列：

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

取🄵於它的建構方式，一個 call 物件會是 (位置引數, 關鍵字引數) 的元組，或是 (名稱, 位置引數, 關鍵字引數) 的元組。當你自己建構它們時，這🄵不是那🄵有趣，但是*Mock.call_args*、*Mock.call_args_list* 和*Mock.mock_calls* 屬性中的 call 物件可以被🄵省以取得它們包含的各個引數。

Mock.call_args 和*Mock.call_args_list* 中的 call 物件是(位置引數, 關鍵字引數)的二元組，而*Mock.mock_calls* 中的 call 物件以及你自己建立的 call 物件是（名稱, 位置引數, 關鍵字引數）的三元組。

你可以利用它們作🄵元組的特性來提取單個引數，以進行更🄵雜的🄵省和斷言。位置引數是一個元組 (如果🄵有位置引數則🄵空元組)，關鍵字引數是一個字典：

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

使用另一個物件作🄵規格建立一個 mock 物件。Mock 上的屬性將使用 *spec* 物件上的對應屬性作🄵其規格。

被 mock 的函式或方法將檢查其引數，以確保他們被使用正確的簽名來呼叫。

如果 *spec_set* 🄵 `True`，則嘗試設定規格物件上不存在的屬性將引發*AttributeError*。

如果一個類🄵作🄵規格使用，則 mock（該類🄵的實例）的回傳值將具有相同的規格。你可以透過傳遞 `instance=True` 來使用一個類🄵作🄵一個實例物件的規格。只有當 mock 的實例是可呼叫物件時，回傳的 mock 才會是可呼叫物件。

create_autospec() 也接受任意的關鍵字引數，這些引數會傳遞給已建立的 mock 的建構函式。

請參🄵*Autospeccing*（自動規格）以得知如何以*create_autospec()* 使用自動規格以及如何在*patch()* 中使用 *autospec* 引數的範例。

在 3.8 版的變更: 如果目標是一個非同步函式，*create_autospec()* 現在會回傳一個*AsyncMock*。

ANY

unittest.mock.**ANY**

有時你可能需要對 mock 的呼叫中的某些引數進行斷言，但你不在意其他的某些引數，或想將它們單獨從*call_args* 中取出🄵進行更加🄵雜的斷言。

要忽略某些引數，你可以傳入對所有物件來🄵都相等的物件。那🄵無論傳入什🄵🄵容，對 `assert_used_with()` 和 `assert_used_once_with()` 的呼叫都會成功。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY 也可以用來與呼叫串列進行比較，例如*mock_calls*：

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

ANY 不只能與呼叫物件比較，其也可以在測試斷言中使用：

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR 是一個模組級🄵的變數，用於控制 mock 物件回應*dir()* 的方式。其預設值🄵 `True`，它使用以下描述的過濾方式來只顯示有用的成員。如果你不喜歡這個過濾方式，或由於診斷意圖而需要將其關閉，請設定 `mock.FILTER_DIR = False`。

當過濾方式開🄵時，`dir(some_mock)` 僅會顯示有用的屬性，🄵將包括通常不會顯示的任何動態建立的屬性。如果 mock 是使用 *spec*（或 *autospec*）來建立的，那🄵源頭的所有屬性都會顯示，即使它們尚未被存取：

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

許多不是很有用的（對*Mock* 來🄵是私有的，而不是被 mock 的東西）底🄵和雙底🄵前綴屬性已從在*Mock* 上呼叫*dir()* 的結果中濾除。如果你不喜歡這種特性，可以透過設定模組級🄵開關*FILTER_DIR* 來將其關閉：

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

或者，你可以只使用 `vars(my_mock)`（實例成員）和 `dir(type(my_mock))`（型🄵成員）來略過過濾，而不考慮*FILTER_DIR*。

mock_open

unittest.mock.**mock_open**(*mock=None*, *read_data=None*)

用於建立取代*open()* 用途的 mock 的輔助函式。它適用於直接呼叫或用作情境管理器的*open()*。

mock 引數是要配置的 mock 物件。如果其🄵 `None`（預設值），那🄵就會🄵你建立一個*MagicMock*，其 API 限制在標準檔案處理上可用的方法或屬性。

read_data 是檔案處理方法*read()*、*readline()* 和*readlines()* 的回傳字串。對這些方法的呼叫將從 *read_data* 取得資料，直到資料耗盡。對這些方法的 mock 非常單純：每次呼叫 *mock* 時，*read_data* 都會倒回到起點。如果你需要對提供給測試程式碼的資料進行更多控制，你會需要自行客🄵化這個 mock。如果這樣還不🄵，PyPI 上的其中一個記憶體🄵檔案系統 (in-memory filesystem) 套件可以提供用於測試的真實檔案系統。

在 3.4 版的變更: 新增對*readline()* 和*readlines()* 的支援。*read()* 的 mock 更改🄵消耗 *read_data* 而不是在每次呼叫時回傳它。

在 3.5 版的變更: 現在，每次呼叫 *mock* 時都會重置 *read_data*。

在 3.8 版的變更: 新增*__iter__()* 到實作中，以便使🄵代（例如在 for 🄵圈中）正確地消耗 *read_data*。

使用*open()* 作🄵情境管理器是確保檔案處理正確關閉的好方式，且這種方式正在變得普遍：

```
with open('/some/path', 'w') as f:
    f.write('something')
```

問題是，即使你 mock 了對*open()* 的呼叫，它也是作🄵情境管理器使用的回傳物件（且其 __enter__() 和 __exit__() 已被呼叫）。

使用*MagicMock* mock 情境管理器相當常見🄵且精細，因此輔助函式就非常有用：

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

以及讀取檔案：

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing（自動規格）

自動規格以 mock 現有的 spec 功能作🄵基礎。它將 mock 的 api 限制🄵原始物件（規格）的 api，但它是遞🄵的（惰性 (lazily) 實現），因此 mock 的屬性僅具有與規格的屬性相同的 api。此外，被 mock 的函式/方法具有與原始的函式/方法相同的呼叫簽名，因此如果它們被不正確地呼叫，就會引發*TypeError*。

在解釋自動規格如何運作之前，我們先解釋🄵什🄵需要它。

Mock 是一個非常🄵大且靈活的物件，但它有一個常見的 mock 缺陷。如果你重構某些程式碼或重新命名成員等，則任何仍然使用舊 *api* 但使用 mock 而非真實物件的程式碼測試仍然會通過。這意味著即使你的程式碼壞了，但測試仍可以全部通過。

在 3.5 版的變更: 在 3.5 之前，當測試應該引發錯誤時，斷言單字中存在拼字錯誤的測驗會默默地通過。你仍可以透過將 unsafe=True 傳遞給 Mock 來實作此行🄵。

謹記這是你需要有整合測試和單元測試的另一個原因。單獨測試所有🄵容都很好，但如果你不測試你的單元是如何「連接在一起」的，那🄵測試還是有機會發現很多錯誤。

unittest.mock 已經提供了一個功能來幫助解🄵這個問題，其稱🄵 speccing。如果你使用類🄵或實例作🄵 mock 的 spec，那🄵你在 mock 上只能存取真實類🄵中存在的屬性：

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

該規格僅適用於 mock 本身，因此在 mock 上的任何方法仍然有相同的問題：

```
>>> mock.header_items()
<mock.Mock object at 0x...>
>>> mock.header_items.assret_called_with()  # 故意的錯字!
```

自動規格解🄵了這個問題。你可以將 autospec=True 傳遞給*patch()* / *patch.object()* 或使用*create_autospec()* 函式建立帶有規格的 mock。如果你對*patch()* 使用 autospec=True 引數，則被

取代的物件將作🄵規格物件使用。因🄵規格是「惰性」完成的（規格是在 mock 被存取時作🄵屬性被建立的），所以你可以將它與非常🄵雜或深度巢狀使用的物件（例如連續引用的模組）一起使用，而不會過於影響性能。

這是一個正在使用的例子：

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

你可以看到 `request.Request` 有一個規格。`request.Request` 在建構函式中接受兩個引數（其中之一是 *self*）。如果我們錯誤地呼叫它，會發生以下情🄵：

```
>>> req = request.Request()
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

此規格也適用於實例化的類🄵（即有規格的 mock 的回傳值）：

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` 物件不是可呼叫物件，因此實例化我們 mock out 的 `request.Request` 的回傳值是不可呼叫的 mock。規格到位後，斷言中的任何拼字錯誤都會引發正確的錯誤：

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

在許多情🄵下，你只需要將 `autospec=True` 新增至現有的*patch()* 呼叫中，然後就可以防止因拼字錯誤和 api 變更而導致的錯誤。

除了透過*patch()* 使用 *autospec* 之外，還有一個*create_autospec()* 用於直接建立有自動規格的 mock：

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

然而，這🄵非完全🄵有限制，這就是🄵什🄵它不是預設的行🄵。🄵了理解規格物件上有哪些可用屬性，autospec 必須🄵省（存取屬性）規格。當你遍歷 mock 上的屬性時，原始物件的對應遍歷正在默默發生。如果你的規格物件具有可以觸發程式碼執行的屬性或描述器，那🄵你可能無法使用 autospec。🄵句話🄵，設計你的物件讓🄵省是安全的[4] 會比較好。

一個更嚴重的問題是，在 `__init__()` 方法中建立實例屬性是常見的，而其根本不存在於類🄵中。*autospec* 無法知道任何動態建立的屬性，🄵將 api 限制🄵可見的屬性。：

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
```

(繼續下一頁)

[4] 這只適用於類🄵或已經實例化的物件。呼叫一個被 mock 的類🄵來建立一個 mock 實例不會建立真的實例。它僅查找屬性及對*dir()* 的呼叫。

(繼續上一頁)

```
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

有幾種不同的方法可以解🄵這個問題。最簡單但可能有點煩人的方法是在建立後簡單地在 mock 上設定所需的屬性。因🄵雖然 *autospec* 不允許你取得規格中不存在的屬性，但是它不會阻止你設定它們：

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

spec 和 *autospec* 有一個更激進的版本，它會*確實*地阻止你設定不存在的屬性。如果你想確保你的程式碼僅能設定有效的屬性，那🄵這會很有用，但顯然它也順便阻止了這個特殊情🄵：

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

解🄵問題的最佳方法可能是新增類🄵屬性作🄵在 `__init__()` 中初始化的實例成員的預設值。請注意，如果你僅在 `__init__()` 中設定預設屬性，那🄵透過類🄵屬性（當然在實例之間共用）提供它們也會更快。例如：

```
class Something:
    a = 33
```

這就帶來了另一個問題。🄵稍後將成🄵不同型🄵的物件的成員提供預設值 `None` 是相對常見的。`None` 作🄵規格是無用的，因🄵它不允許你存取其上的*任何*屬性或方法。由於 `None` 作🄵規格永遠不會有用，🄵且可能表示通常屬於其他型🄵的成員，因此自動規格不會對設定🄵 `None` 的成員使用規格。這些會只是普通的 mock（通常是 MagicMocks）：

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

如果修改正式生🄵 (production) 類🄵以新增預設值不符合你的喜好，那🄵還有更多選擇。其中之一就是簡單地使用實例作🄵規格而不是使用類🄵。另一種是建立一個正式生🄵類🄵的子類🄵，🄵將預設值新增至子類🄵中，而不影響正式生🄵類🄵。這兩個都需要你使用替代物件作🄵規格。值得慶幸的是*patch()* 支援這一點 - 你可以簡單地將替代物件作🄵 *autospec* 引數傳遞：

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
```

(繼續下一頁)

(繼續上一頁)

```
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

密封 mock

unittest.mock.**seal**(*mock*)

當存取被密封的 mock 的屬性或其任何已經遞🄵 mock 的屬性時，seal 將停用 mock 的自動建立。

如果將具有名稱或規格的 mock 實例指派給屬性，則不會出現在密封鏈中。這表示可藉由固定 mock 物件的一部分來防止密封。:

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

在 3.7 版被加入.

26.6.6 `side_effect`、`return_value` 和 *wraps* 的優先順序

它們的優先順序是:

1. *side_effect*
2. *return_value*
3. *wraps*

如果這三個都有設定，mock 將會回傳來自*side_effect* 的值，🄵忽略*return_value* 和被包裝物件。如果設定了任兩項，則優先順序較高的一項將回傳該值。無論先設定哪個順序，優先順序都保持不變。

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

由於 None 是*side_effect* 的預設值，如果將其值重新賦值回🄵 None，則會檢查*return_value* 和被包裝物件之間的優先順序，忽略*side_effect*。

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

如果*side_effect* 回傳的值是*DEFAULT*，它將被忽略，🄵且優先順序被移動到後面一個以獲得要回傳的值。

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

當*Mock* 包裝一個物件時，*return_value* 的預設值將🄴*DEFAULT*。

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

優先順序將忽略該值，🄴將移動到最後一個，即被包裝物件。

當對被包裝物件進行真正的呼叫時，建立此 mock 的實例將回傳該類🄴的真實實例。必須傳遞被包裝物件所需的位置引數（如果存在）。

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

但如果你🄴其賦予 `None` 則不會被忽略，因🄴它是明確賦值。因此，優先順序不會移至被包裝物件。

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

即使你在初始化 mock 時同時設定所有三個，優先順序也保持不變：

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

如果*side_effect* 已耗盡，則優先順序將不會使值由後面取得。相反地這會引發 `StopIteration` 例外。

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect value",
...                                     "another side effect value"]
>>> order_mock.get_value.return_value = "second"
```

```
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'
```

```
>>> order_mock.get_value()
Traceback (most recent call last):
  ...
StopIteration
```

26.7 unittest.mock --- 入門指南

在 3.3 版被加入.

26.7.1 使用 Mock 的方式

使用 Mock 來 patching 方法

Mock 物件的常見用法包含：

- Patching 方法
- 記🄵在物件上的方法呼叫

你可能會想要取代一個物件上的方法，以便檢查系統的另一部分是否使用正確的引數呼叫它：

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

一旦我們的 mock 已經被使用（例如在這個範例中的 `real.method`），它就有了方法和屬性，允許你對其使用方式進行斷言 (assertions)。

> **備🄵**
>
> 在大多數的範例中，*Mock* 和*MagicMock* 類🄵是可以互🄵的。不過由於 `MagicMock` 是功能更🄵大的類🄵，因此通常它是一個更好的選擇。

一旦 mock 被呼叫，它的*called* 屬性將被設定🄵 `True`。更重要的是，我們可以使用*assert_called_with()* 或*assert_called_once_with()* 方法來檢查它是否被使用正確的引數來呼叫。

這個範例測試呼叫 `ProductionClass().method` 是否導致對 `something` 方法的呼叫：

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

對物件的方法呼叫使用 mock

在上一個範例中，我們直接對物件上的方法進行 patch，以檢查它是否被正確呼叫。另一個常見的用法是將一個物件傳遞給一個方法（或受測系統的某一部分），然後檢查它是否以正確的方式被使用。

下面是一個單純的 `ProductionClass`，含有一個 `closer` 方法。如果它被傳入一個物件，它就會呼叫此物件中的 `close`。

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

因此，🄵了對此進行測試，我們需要傳遞一個具有 `close` 方法的物件，🄵檢查它是否被正確的呼叫。

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

我們不必做任何額外的事情來🄵 mock 提供'close' 方法，存取 close 會建立它。因此，如果'close' 🄵未被呼叫過，在測試中存取'close' 就會建立它，但*`assert_called_with()`* 就會引發一個失敗的例外。

Mock 類🄵

一個常見的使用案例是在測試的時候 mock 被程式碼實例化的類🄵。當你 patch 一個類🄵時，該類🄵就會被替🄵🄵 mock。實例是透過呼叫類🄵建立的。這代表你可以透過查看被 mock 的類🄵的回傳值來存取「mock 實例」。

在下面的範例中，我們有一個函式 `some_function`，它實例化 `Foo` 🄵呼叫它的方法。對*`patch()`* 的呼叫將類🄵 `Foo` 替🄵🄵一個 mock。`Foo` 實例是呼叫 mock 的結果，因此它是透過修改 mock *`return_value`* 來配置的。:

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

命名你的 mock

🄵你的 mock 命名可能會很有用。這個名稱會顯示在 mock 的 repr 中，且當 mock 出現在測試的失敗訊息中時，名稱會很有幫助。該名稱也會傳播到 mock 的屬性或方法:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

追🄵所有呼叫

通常你會想要追🄵對一個方法的多個呼叫。*`mock_calls`* 屬性記🄵對 mock 的子屬性以及其子屬性的所有呼叫。

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

如果你對 `mock_calls` 做出斷言🄵且有任何不預期的方法被呼叫，則斷言將失敗。這很有用，因🄵除了斷言你期望的呼叫已經進行之外，你還可以檢查它們是否按正確的順序進行，🄵且🄵有多余的呼叫:

你可以使用*call* 物件來建構串列以與 `mock_calls` 進行比較：

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

然而，回傳 mock 的呼叫的參數不會被記🄵，這代表在巢狀呼叫中，無法追🄵用於建立上代的參數 important 的值：

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

設定回傳值和屬性

在 mock 物件上設定回傳值非常簡單：

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

當然，你可以對 mock 上的方法執行相同的操作：

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

回傳值也可以在建構函式中設定：

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

如果你需要在 mock 上進行屬性設置，只需執行以下操作：

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

有時你想要 mock 更🄵雜的情🄵，例如 `mock.connection.cursor().execute("SELECT 1")`。如果我們希望此呼叫回傳一個串列，那🄵我們就必須配置巢狀呼叫的結果。

如下所示，我們可以使用*call* 在「🄵接呼叫 (chained call)」中建構呼叫的集合，以便在之後輕松的進行斷言：

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

正是對 `.call_list()` 的呼叫將我們的呼叫物件轉🄵🄵代表🄵接呼叫的呼叫串列。

透過 mock 引發例外

一個有用的屬性是*side_effect*。如果將其設定🄵例外類🄵或實例，則當 mock 被呼叫時將引發例外。

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect 函式以及可🄵代物件

`side_effect` 也可以設定🄵函式或可🄵代物件。`side_effect` 作🄵可🄵代物件的使用案例是：當你的 mock 將會被多次呼叫，且你希望每次呼叫回傳不同的值。當你將 `side_effect` 設定🄵可🄵代物件時，對 mock 的每次呼叫都會傳回可🄵代物件中的下一個值：

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

對於更進階的使用案例，例如根據 mock 被呼叫的🄵容動態變更回傳值，可以將 `side_effect` 設成一個函式。該函式會使用與 mock 相同的引數被呼叫。函式回傳的🄵容就會是呼叫回傳的🄵容：

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Mock 非同步可🄵代物件

從 Python 3.8 開始，`AsyncMock` 和 `MagicMock` 支援透過 `__aiter__` 來 mock async-iterators。`__aiter__` 的*return_value* 屬性可用來設定用於🄵代的回傳值。

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

Mock 非同步情境管理器

從 Python 3.8 開始，`AsyncMock` 和 `MagicMock` 支援透過 `__aenter__` 和 `__aexit__` 來 mock async-context-managers。預設情🄵下，`__aenter__` 和 `__aexit__` 是回傳非同步函式的 `AsyncMock` 實例。

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

從現有物件建立 mock

過度使用 mock 的一個問題是，它將你的測試與 mock 的實作結合在一起，而不是與真實的程式碼結合。假設你有一個實作 `some_method` 的類🄵，在另一個類🄵的測試中，你提供了一個 mock 的物件，其也提供了 `some_method`。如果之後你重構第一個類🄵，使其不再具有 `some_method` - 那🄵即使你的程式碼已經損壞，你的測試也將繼續通過！

Mock 允許你使用 *spec* 關鍵字引數提供一個物件作🄵 mock 的規格。對 mock 存取規格物件上不存在的方法或屬性將立即引發一個屬性錯誤。如果你更改規格的實作，那🄵使用該類🄵的測試將立即失敗，而無需在這些測試中實例化該類🄵。

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did you mean: 'class_method'?
```

使用規格還可以更聰明地匹配對 mock 的呼叫，無論引數是作🄵位置引數還是命名引數傳遞：

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

如果你希望這種更聰明的匹配也可以應用於 mock 上的方法呼叫，你可以使用自動規格。

如果你想要一種更🄵大的規格形式來防止設定任意屬性以及取得它們，那🄵你可以使用 *spec_set* 而不是 *spec*。

使用 side_effect 回傳各🄵檔案🄵容

mock_open() 是用於 patch *open()* 方法。*side_effect* 可以用來在每次呼叫回傳一個新的 mock 物件。這可以用於回傳儲存在字典中的各🄵檔案的不同🄵容：

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"

    with open("file3") as file2:
        assert file2.read() == "default"
```

26.7.2 Patch 裝飾器

> **備🄵**
>
> 使用*patch()* 時，需注意的是你得在被查找物件的命名空間中（in the namespace where they are looked up）patch 物件。這通常很直接，但若需要快速導引，請參🄵*該 patch 何處*。

測試中的常見需求是 patch 類🄵屬性或模組屬性，例如 patch 一個🄵建函式（built-in）或 patch 模組中的類🄴以測試它是否已實例化。模組和類🄴實際上是全域的，因此在測試後必須撤銷它們的 patch，否則 patch 將延續到其他測試中🄵導致難以診斷問題。

mock 🄵此提供了三個方便的裝飾器：*patch()*、*patch.object()* 和*patch.dict()*。patch 接受單一字串，格式🄵 `package.module.Class.attribute`，用來指定要 patch 的屬性。同時它也可以接受你想要替🄴的屬性（或類🄴或其他）的值。`patch.object` 接受一個物件和你想要 patch 的屬性的名稱，同時也可以接受要 patch 的值。

`patch.object`：

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

如果你要 patch 一個模組（包括*builtins*），請使用*patch()* 而非*patch.object()*：

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

如有需要，模組名稱可以含有 `.`，形式🄵 `package.module`：

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

一個好的模式是實際裝飾測試方法本身：

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

如果你想使用一個 mock 進行 patch，你可以使用僅帶有一個引數的*patch()*（或帶有兩個引數的*patch.object()*）。Mock 將被建立F被傳遞到測試函式 / 方法中：

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

你可以使用這個模式堆F多個 patch 裝飾器：

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

當你巢狀使用 patch 裝飾器時，mock 傳遞到被裝飾函式的順序會跟其被應用的順序相同（一般 *Python* 應用裝飾器的順序）。這意味著由下而上，因此在上面的範例中，`module.ClassName2` 的 mock 會先被傳入。

也有*patch.dict()*，用於在測試範圍中設定字典F的值，F在測試結束時將其恢復F原始狀態：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`、`patch.object` 和 `patch.dict` 都可以用來作F情境管理器。

當你使用*patch()* F你建立一個 mock 時，你可以使用 with 陳述式的"as" 形式來取得 mock 的參照：

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

另外，`patch`、`patch.object` 和 `patch.dict` 也可以用來作F類F裝飾器。以這種方式使用時，與將裝飾器單獨應用於每個名稱以"test" 開頭的方法相同。

26.7.3 更多范例

以下是一些更進階一點的情境的範例。

Mock F接呼叫

一旦你了解了*return_value*屬性，mock F接呼叫其實就很簡單了。當一個 mock 第一次被呼叫，或者你在它被呼叫之前取得其 `return_value` 時，一個新的*Mock*就會被建立。

這代表你可以透過查問 `return_value` mock 來了解一個對被 mock 的物件的呼叫回傳的物件是如何被使用的：

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

這🄵只需一個簡單的步驟即可進行配置🄵對🄵接呼叫進行斷言。當然，另一種選擇是先以更容易被測試的方式撰寫程式碼...

所以，假設我們有一些程式碼，看起來大概像這樣：

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs').start_
↪call()
...         # more code
```

假設 `BackendProvider` 已經經過充分測試，那🄵我們該如何測試 `method()`？具體來🄵，我們要測試程式碼部分 `# more code` 是否以正確的方式使用 `response` 物件。

由於此呼叫🄵是從實例屬性進行的，因此我們可以在 `Something` 實例上 monkey patch `backend` 屬性。在這種特定的情🄵下，我們只對最終呼叫 `start_call` 的回傳值感興趣，因此我們不需要做太多配置。我們假設它傳回的物件是類檔案物件 (file-like)，因此我們會確保我們的 response 物件使用🄵建的*`open()`* 作🄵其 `spec`。

🄵此，我們建立一個 mock 實例作🄵我們的 mock backend，🄵🄵其建立一個 mock response 物件。要將 response 設定🄵最後的 `start_call` 的回傳值，我們可以這樣做：

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_value =␣
↪mock_response
```

我們可以使用*`configure_mock()`* 方法來以稍微好一點的方式🄵我們直接設定回傳值：

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_value':␣
↪mock_response}
>>> mock_backend.configure_mock(**config)
```

有了這些，我們就可以原地 (in place) monkey patch "mock backend"，🄵且可以進行真正的呼叫：

```
>>> something.backend = mock_backend
>>> something.method()
```

藉由使用*`mock_calls`*，我們可以使用單一一個斷言來檢查🄵接呼叫。一個🄵接呼叫是一行程式碼中的多個呼叫，因此 `mock_calls` 中會有多個項目。我們可以使用*`call.call_list()`* 來🄵我們建立這個呼叫串列：

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

部分 mocking

在某些測試中，我們會想 mock 對*`datetime.date.today()`* 的呼叫以回傳一個已知日期，但我不想阻止測試中的程式碼建立新的日期物件。不幸的是*`datetime.date`* 是用 C 語言寫的，所以們我不能 monkey patch 🄵態的*`datetime.date.today()`* 方法。

我們找到了一種簡單的方法來做到這一點，其用 mock 有效地包裝日期類🄵，但將對建構函式的呼叫傳遞給真實的類🄵（🄵返回真實的實例）。

這個使用 *patch 裝飾器* 來 mock 被測模組中的 `date` 類別。然後，mock 日期類別上的 *side_effect* 屬性會被設定為回傳真實日期的 lambda 函式。當 mock 日期類別被呼叫時，將透過 `side_effect` 建構並回傳真實日期。:

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

注意，我們有全域 patch *datetime.date*，而是在使用它的模組中 patch `date`。請參閱*該 patch 何處*。

當 `date.today()` 被呼叫時，一個已知日期會被回傳，但對 `date(...)` 建構函式的呼叫仍然會回傳正常日期。如果不這樣使用，你可能會發現自己必須使用與被測程式碼完全相同的演算法來計算預期結果，這是一個典型的測試的反面模式 (anti-pattern)。

對日期建構函式的呼叫被記在 `mock_date` 屬性（`call_count` 及其相關屬性）中，這對你的測試也可能有用處。

處理 mock 日期或其他建類別的另一種方法在 這個 blog 中討論。

Mock 生器方法

Python 生器是一個函式或方法，它使用 `yield` 陳述式在代[1] 時回傳一系列的值。

生器方法 / 函式會被呼叫以回傳生器物件。之後此生器會進行代。代的協定方法是 *__iter__()*，所以我們可以使用 *MagicMock* 來 mock 它。

下面是一個範例類別，其包含實作生器的一個"iter" 方法:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

我們該如何 mock 這個類別，特是它的"iter" 方法呢?

要配置從代回傳的值（隱含在對 *list* 的呼叫中），我們需要配置呼叫 `foo.iter()` 所回傳的物件。

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

對每個測試方法應用相同的 patch

如果你希望 patch 能用在多個測試方法上，顯而易見的方式是將 patch 裝飾器應用於每個方法。這感覺是不必要的重行，因此你可以使用 *patch()*（及其他 patch 的變體）作類裝飾器。這會將 patch 應用在該類的所有測試方法上。測試方法由名稱以 `test` 開頭來識:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
```

(繼續下一頁)

[1] 還有關於生器運算式及生器的更多 進階用法，但我們在這不考慮它們。一個關於生器及其大功能的優良明是: Generator Tricks for Systems Programmers。

(繼續上一頁)

```
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

管理 patch 的另一種方式是使用*patch 方法：🄵動與停止*。這允許你將 patch 移到你的 `setUp` 與 `tearDown` 方法中。:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

如果你使用這個技巧，你必須確保透過呼叫 `stop` 來" 取消" patch。這可能會比你想像的還要🄵雜一點，因🄵如果有例外在 `setUp` 中被引發，則 `tearDown` 就不會被呼叫。*`unittest.TestCase.addCleanup()`* 會讓這稍微簡單一點:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mock Unbound Methods （未🄵結方法）

在撰寫測試時，當我們需要 patch 一個未🄵結方法（patch 類🄵上的方法而不是實例上的方法）。我們需要將 self 作🄵第一個引數傳入，因🄵我們想斷言哪些物件正在呼叫這個特定方法。問題是你無法🄵此使用 mock 進行 patch，因🄵就算你用一個 mock 替🄵未🄵結方法，從實例取得它時它也不會成🄵一個🄵結方法，因此 self 🄵不會被傳遞。解🄵方法是使用真實的函式來 patch 未🄵結方法。*`patch()`* 裝飾器使得用 mock 來 patch out 方法是如此的簡單，以至於建立一個真正的函式相對變得很麻煩。

如果你將 `autospec=True` 傳遞給 patch，那🄵它會使用真的函式物件來進行 patch。此函式物件與它所替🄵的函式物件具有相同的簽名，但實際上委🄵給 mock。你仍然會以與之前完全相同的方式自動建立 mock。但這意味著，如果你使用它來 patch 類🄵上的未🄵結方法，則從實例取得的 mock 函式將轉🄵🄵🄵結方法。`self` 會作🄵其第一個引數傳入，而這正是我們想要的:

```
>>> class Foo:
...   def foo(self):
...     pass
...
```

(繼續下一頁)

(繼續上一頁)

```
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

如果我們不使用 `autospec=True`，那🄵未🄵結方法將使用一個 Mock 實例 patch out，🄵且不被使用 `self` 進行呼叫。

使用 mock 檢查多個呼叫

mock 有很好的 API，用於對 mock 物件的使用方式做出斷言。

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

如果你的 mock 只被呼叫一次，你可以使用*assert_called_once_with()* 方法，其也斷言*call_count* 是 1。

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected 'foo_bar' to be called once. Called 2 times.
Calls: [call('baz', spam='eggs'), call()].
```

`assert_called_with` 和 `assert_called_once_with` 都對最近一次的呼叫做出斷言。如果你的 mock 將被多次呼叫，🄵且你想要對所有這些呼叫進行斷言，你可以使用*call_args_list*：

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

call 輔助函式可以輕松地對這些呼叫做出斷言。你可以建立預期呼叫的清單🄵將其與 `call_args_list` 進行比較。這看起來與 `call_args_list` 的 repr 非常相似：

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

應對可變引數

另一種情🄵很少見，但可能會困擾你，那就是當你的 mock 被使用可變引數呼叫。`call_args` 和 `call_args_list` 儲存對引數的參照。如果引數被測試中的程式碼改變，那🄵你將無法再對 mock 被呼叫時的值進行斷言。

這是一些秀出問題的程式碼範例。想像'mymodule' 中定義以下函式：

```
def frob(val):
    pass

def grob(val):
```

(繼續下一頁)

(繼續上一頁)

```
    "First frob and then clear val"
    frob(val)
    val.clear()
```

當我們嘗試測試 grob 使用正確的引數呼叫 frob 時，看看會發生什🄵:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

一種可能是讓 mock 🄵🄵你傳入的引數。如果你進行的斷言依賴於物件識🄴性來確定相等性，這就可能導致問題。

以下是一種使用*side_effect* 功能的解法。如果你🄵 mock 提供一個 side_effect 函式，則 side_effect 將被使用與 mock 相同的引數呼叫。這使我們有機會🄵🄵引數🄵將其儲存以供之後的斷言。在這個範例中，我們使用另一個 mock 來儲存引數，以便我們可以使用 mock 方法來執行斷言。同樣的，有一個輔助函式🄵我們設定了這些。:

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

copy_call_args 與將要被呼叫的 mock 一起被呼叫。它回傳一個我們會對其進行斷言的新的 mock。side_effect 函式建立引數們的副本，🄵用該副本呼叫我們的 new_mock。

備🄵

如果你的 mock 只會被使用一次，則有一種更簡單的方法可以在呼叫引數時檢查它們。你可以簡單地在 side_effect 函式🄴進行檢查。

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
```

```
Traceback (most recent call last):
    ...
AssertionError
```

另一種方法是建立*Mock* 或*MagicMock* 的子類F來FF（使用*copy.deepcopy()*）引數。這是一個實作的例子：

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

當你將 `Mock` 或 `MagicMock` 子類F化時，所有屬性會被動態建立，且 `return_value` 會自動使用你的子類F。這代表著 `CopyingMock` 的所有子代 (child) 也會具有 `CopyingMock` 型F。

巢狀使用 Patch

將 patch 作F情境管理器使用很好，但是如果你使用F數個 patch，你最終可能會得到巢狀的 with 陳述式，F且越來越向右縮排：

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

我們可以使用 unittest 的 `cleanup` 函式以及*patch 方法：F動與停止* 來達到相同的效果，而不會出現因巢狀導致的縮排。一個簡單的輔助方法 `create_patch` 會將 patch 放置到位FF我們回傳被建立的 mock：

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
```

(繼續下一頁)

(繼續上一頁)

```
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

使用 MagicMock 來 mock 字典

你可能會想要 mock 字典或其他容器物件，記🄵對它的所有存取，同時讓它仍然像字典一樣運作。

我們可以使用*MagicMock* 來做到這一點，它的行🄵會與字典一致，🄵使用*side_effect* 將字典存取委🄵給我們控制下的真實底層字典。

當 MagicMock 的 __getitem__() 和 __setitem__() 方法被呼叫時（一般的字典存取），side_effect 會被使用鍵 (key) 來呼叫（在 __setitem__ 的情🄵也會使用值 (value)）。我們也可以控制回傳的🄵容。

使用 MagicMock 後，我們可以使用諸如*call_args_list* 之類的屬性來斷言字典被使用的方式：

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

備🄵

不使用 MagicMock 的替代方案是使用 Mock 🄵且只提供你特🄵想要的魔術方法：

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

第三個選擇是使用 MagicMock，但傳入 dict 作🄵 *spec*（或 *spec_set*）引數，以使被建立的 MagicMock 僅具有字典可用的魔術方法：

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

有了這些 side effect 函式，mock 會像一般的字典一樣運作，但會記🄵存取。如果你嘗試存取不存在的鍵，它甚至會引發一個*KeyError*。

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
```

(繼續下一頁)

(繼續上一頁)

```
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

在上述方式被使用後，你就可以使用普通的 mock 方法和屬性對存取進行斷言：

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock 子類🄵及其屬性

你會想子類🄵化*Mock* 的原因可能有很多種。其中之一是增加輔助方法。以下是一個有點笨的例子：

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

`Mock` 實例的標准行🄵是屬性 mock 和回傳值 mock 會與存取它們的 mock 具有相同的型🄵。這確保了 `Mock` 屬性是 `Mocks`，`MagicMock` 屬性是 `MagicMocks`[2]。因此，如果你要子類🄵化以新增輔助方法，那🄵它們也可用於子類🄵實例的屬性 mock 和回傳值 mock。

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

有時候這很不方便。例如，有一個使用者正在子類🄵化 mock 以建立 Twisted adaptor。將其應用於屬性實際上會導致錯誤。

`Mock`（及其各種形式）使用名🄵 `_get_child_mock` 的方法來🄵屬性和回傳值建立這些" 子 mock"。你可以透過置🄵此方法來防止你的子類🄵被用🄵屬性。其簽名是取用任意的關鍵字引數（`**kwargs`），然後將其傳遞給 mock 建構函式：

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
```

(繼續下一頁)

[2] 此規則的例外是非可呼叫物件的 mock。屬性使用可呼叫物件的變體，否則非可呼叫物件的 mock 無法具有可呼叫的方法。

(繼續上一頁)

```
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

使用 patch.dict 來 mock import

可能會讓 mock 很困難的一種情🄵是在函式🄵部進行區域 import。這些狀🄵會更難進行 mock，因🄵它們🄵有使用我們可以 patch out 的模組命名空間中的物件。

一般來🄵，我們應該避免區域 import 的發生。有時這樣做是🄵了防止循環相依 (circular dependencies)，🄵此通常有更好的方式來解🄵問題（例如重構程式碼）或透過延遲 import 來防止「前期成本 (up front costs)」。這也可以透過比無條件的區域 import 更好的方式來解🄵（例如將模組儲存🄵類🄵或模組屬性，🄵且僅在第一次使用時進行 import）。

除此之外，還有一種方法可以使用 `mock` 來影響 import 的結果。Import 會從*`sys.modules`* 字典中取得一個物件。請注意，它會取得一個物件，而該物件不需要是一個模組。初次 import 模組會導致模組物件被放入 `sys.modules` 中，因此通常當你 import 某些東西時，你會得到一個模組。但🄵非一定要如此。

這代表你可以使用*`patch.dict()`* 來暫時在*`sys.modules`* 中原地放置 mock。在這個 patch 作用時的任何 import 都會取得這個 mock。當 patch 完成時（被裝飾的函式結束、with 陳述式主體完成或 `patcher.stop()` 被呼叫），那🄵之前在 `sys.modules` 中的任何🄵容都會被安全地復原。

下面是一個 mock out 'fooble' 模組的例子。

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

如你所見，`import fooble` 成功了，但在離開之後，*`sys.modules`* 中就🄵有'fooble' 了。

這也適用於 `from module import name` 形式：

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

透過稍微多一點的處理，你也可以 mock 套件的引入：

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

追🄵呼叫順序與更簡潔的呼叫斷言

Mock 類🄵可以讓你透過*method_calls* 屬性追🄵 mock 物件上方法呼叫的順序。這不會讓你可以追🄵不同的 mock 物件之間的呼叫順序，然而我們可以使用*mock_calls* 來達到相同的效果。

因🄵 mock 會在 mock_calls 中追🄵對子 mock 的呼叫，且存取 mock 的任意屬性會建立一個子 mock，所以我們可以從父 mock 建立不同的 mock。之後對這些子 mock 的呼叫將依序全部記🄵在父 mock 的 mock_calls 中：

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

之後我們可以透過與管理器 (manager) mock 上的 mock_calls 屬性進行比較來斷言呼叫及其順序：

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

如果 patch 正在被建立🄵放置🄵你的 mock，那🄵你可以使用*attach_mock()* 方法將它們附加到管理器 mock。附加之後，呼叫將被記🄵在管理器的 mock_calls 中。：

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

如果進行了多次呼叫，但你只對其中特定呼叫的序列感興趣，則可以使用*assert_has_calls()* 方法。這需要一個呼叫串列（使用*call* 物件建構）。如果該呼叫序列位於*mock_calls* 中，則斷言成功。

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

🄵管🄵接呼叫 m.one().two().three() 🄵不是對 mock 進行的唯一呼叫，斷言仍會成功。

有時可能會對一個 mock 進行多次呼叫，而你只對斷言其中某些呼叫感興趣。你甚至可能不關心順序。在這種情🄵下，你可以將 any_order=True 傳遞給 assert_has_calls：

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

更🄵雜的引數匹配

使用與*ANY* 相同的基本概念，我們可以實作匹配器 (matcher)，對用來作🄵 mock 引數的物件進行更🄵雜的斷言。

假設我們預期某個物件會被傳進 mock，預設情🄵下，該 mock 會根據物件識🄵性進行相等比較（對使用者定義類🄵的 Python 預設行🄵）。要使用*assert_called_with()*，我們需要傳入完全相同的物件。如果我們只對該物件的某些屬性感興趣，那🄵我們可以建立一個匹配器來🄵我們檢查這些屬性。

你可以在這個範例中看到對 `assert_called_with` 的一個「標準」呼叫是不🄵的：

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

對我們的 `Foo` 類🄵的比較函式看起來可能會像這樣：

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

而可以使用像這樣的比較函式進行其相等性運算的匹配器物件會長得像這樣：

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

把這些都放在一起：

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

`Matcher` 是用我們想要比較的 `Foo` 物件和比較函式實例化的。在 `assert_called_with` 中，`Matcher` 相等方法將被呼叫，它將呼叫 mock 時傳入的物件與我們建立匹配器時傳入的物件進行比較。如果它們匹配，則 `assert_called_with` 會通過，如果它們不匹配，則會引發*AssertionError*：

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

透過一些調整，你可以讓比較函式直接引發*AssertionError* 🄵提供更有用的失敗訊息。

從版本 1.5 開始，Python 測試函式庫 PyHamcrest 以其相等匹配器的形式，提供了類似的功能，在這🄵可能是有用的 (hamcrest.library.integration.match_equality)。

26.8 `test` --- Python 的回歸測試 (regression tests) 套件

備🄵

The *test* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *test* package contains all regression tests for Python as well as the modules *test.support* and *test.regrtest*. *test.support* is used to enhance your tests while *test.regrtest* drives the testing suite.

Each module in the *test* package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

也參考

***unittest* 模組**
撰寫 PyUnit 回歸測試。

***doctest* 模組**
鑲嵌在文件字串中的測試。

26.8.1 撰寫 `test` 套件的單元測試

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
```

(繼續下一頁)

(繼續上一頁)

```
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by *test.regrtest*, on its own as a script that supports the *unittest* CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also ”private” code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]
```

(繼續下一頁)

(繼續上一頁)

```
class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

When using this pattern, remember that all classes that inherit from *unittest.TestCase* are run as tests. The `TestFuncAcceptsSequencesMixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from *unittest.TestCase*.

也參考

測試驅動開發
由 Kent Beck 所著，關於先寫測試再寫程式的書籍。

26.8.2 使用命令列介面執行測試

The *test* package can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **`python -m test`**. Under the hood, it uses *test.regrtest*; the call **`python -m test.regrtest`** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the *test* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**`python -m test test_spam`**) will minimize output and only print whether the test passed or failed.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **`python -m test -uall`**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **`python -m test -uall,-audio,-largefile`** will run *test* with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **`python -m test -h`**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **`make test`** at the top-level directory where Python was built. On Windows, executing **`rt.bat`** from your `PCbuild` directory will run all regression tests.

在 3.14 版被加入: Output is colorized by default and can be controlled using environment variables.

26.9 `test.support` --- Python 測試套件的工具

test.support 模組提供 Python 回歸測試套件的支援。

備🄵

test.support is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

此模組定義了以下例外：

exception `test.support.`**TestFailed**

Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception `test.support.`**ResourceDenied**

Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

test.support 模組定義了以下常數：

test.support.**verbose**

`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by *test.regrtest*.

test.support.**is_jython**

如果執行的直譯器是 Jython，則🄵 `True`。

test.support.**is_android**

如果 `sys.platform` 是 `android` 則🄵 `True`。

test.support.**is_emscripten**

如果 `sys.platform` 是 `emscripten` 則🄵 `True`。

test.support.**is_wasi**

如果 `sys.platform` 是 `wasi` 則🄵 `True`。

test.support.**is_apple_mobile**

如果 `sys.platform` 是 `ios`、`tvos` 或 `watchos` 則🄵 `True`。

test.support.**is_apple**

`True` if `sys.platform` is `darwin` or `is_apple_mobile` is `True`.

test.support.**unix_shell**

Path for shell if not on Windows; otherwise `None`.

test.support.**LOOPBACK_TIMEOUT**

Timeout in seconds for tests using a network server listening on the network local loopback interface like `127.0.0.1`.

The timeout is long enough to prevent test failure: it takes into account that the client and the server can run in different threads or even different processes.

The timeout should be long enough for *connect()*, *recv()* and *send()* methods of *socket.socket*.

預設值🄵 5 秒。

另請參🄵*INTERNET_TIMEOUT*。

test.support.**INTERNET_TIMEOUT**

Timeout in seconds for network requests going to the internet.

The timeout is short enough to prevent a test to wait for too long if the internet request is blocked for whatever reason.

Usually, a timeout using *INTERNET_TIMEOUT* should not mark a test as failed, but skip the test instead: see *transient_internet()*.

預設值🄵 1 分鐘。

另請參🄵*LOOPBACK_TIMEOUT*。

test.support.**SHORT_TIMEOUT**

Timeout in seconds to mark a test as failed if the test takes ”too long”.

The timeout value depends on the regrtest `--timeout` command line option.

If a test using *SHORT_TIMEOUT* starts to fail randomly on slow buildbots, use *LONG_TIMEOUT* instead.

預設值🄵 30 秒。

test.support.**LONG_TIMEOUT**

Timeout in seconds to detect when a test hangs.

It is long enough to reduce the risk of test failure on the slowest Python buildbots. It should not be used to mark a test as failed if the test takes ”too long”. The timeout value depends on the regrtest `--timeout` command line option.

預設值🄵 5 分鐘。

請參🄵*LOOPBACK_TIMEOUT*、*INTERNET_TIMEOUT* 和*SHORT_TIMEOUT*。

test.support.**PGO**

Set when tests can be skipped when they are not useful for PGO.

test.support.**PIPE_MAX_SIZE**

A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.**Py_DEBUG**

`True` if Python was built with the `Py_DEBUG` macro defined, that is, if Python was built in debug mode.

在 3.12 版被加入.

test.support.**SOCK_MAX_SIZE**

A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.**TEST_SUPPORT_DIR**

設定🄵包含*test.support* 的頂層目🄵。

test.support.**TEST_HOME_DIR**

設定🄵測試套件的頂層目🄵。

test.support.**TEST_DATA_DIR**

設定🄵測試套件中的 `data` 目🄵。

test.support.**MAX_Py_ssize_t**

設定🄵*sys.maxsize* 以進行大記憶體測試。

test.support.**max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Limited by *MAX_Py_ssize_t*.

test.support.**real_max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Not limited by *MAX_Py_ssize_t*.

test.support.**MISSING_C_DOCSTRINGS**

Set to `True` if Python is built without docstrings (the `WITH_DOC_STRINGS` macro is not defined). See the `configure --without-doc-strings` option.

另請參🄵*HAVE_DOCSTRINGS* 變數。

test.support.**HAVE_DOCSTRINGS**

Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

請參🄵*MISSING_C_DOCSTRINGS* 變數。

test.support.**TEST_HTTP_URL**

Define the URL of a dedicated HTTP server for the network tests.

test.support.**ALWAYS_EQ**

Object that is equal to anything. Used to test mixed type comparison.

test.support.**NEVER_EQ**

Object that is not equal to anything (even to *ALWAYS_EQ*). Used to test mixed type comparison.

test.support.**LARGEST**

Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**

Object that is less than anything (except itself). Used to test mixed type comparison.

test.support 模組定義了以下函式：

test.support.**busy_retry**(*timeout*, *err_msg=None*, */*, ***, *error=True*)

執行🄵圈主體直到 `break` 停止🄵圈。

After *timeout* seconds, raise an *AssertionError* if *error* is true, or just stop the loop if *error* is false.

範例：

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 用法範例：

```
for _ in support.busy_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**sleeping_retry**(*timeout*, *err_msg=None*, */*, ***, *init_delay=0.010*, *max_delay=1.0*, *error=True*)

Wait strategy that applies exponential backoff.

Run the loop body until `break` stops the loop. Sleep at each loop iteration, but not at the first iteration. The sleep delay is doubled at each iteration (up to *max_delay* seconds).

請見*busy_retry()* 文件以🄵解參數用法。

在 SHORT_TIMEOUT 秒後引發例外的範例：

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 用法範例：

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**is_resource_enabled**(*resource*)

Return `True` if *resource* is enabled and available. The list of available resources is only set when *test.regrtest* is executing the tests.

test.support.**python_is_optimized**()

如果 Python 不是使用 `-O0` 或 `-Og` 建置則回傳 `True`。

test.support.**with_pymalloc**()

回傳 `_testcapi.WITH_PYMALLOC`。

test.support.**requires**(*resource*, *msg=None*)

Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by *test.regrtest*.

test.support.**sortdict**(*dict*)

Return a repr of *dict* with keys sorted.

test.support.**findfile**(*filename*, *subdir=None*)

Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.**get_pagesize**()

Get size of a page in bytes.

在 3.12 版被加入.

test.support.**setswitchinterval**(*interval*)

Set the *sys.setswitchinterval()* to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.**check_impl_detail**(***guards*)

Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments. This function returns `True` or `False` depending on the host platform. Example usage:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**set_memlimit**(*limit*)

Set the values for *max_memuse* and *real_max_memuse* for big memory tests.

test.support.**record_original_stdout**(*stdout*)

Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

test.support.**get_original_stdout**()

Return the original stdout set by *record_original_stdout()* or `sys.stdout` if it's not set.

test.support.**args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

test.support.**optim_args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

A context managers that temporarily replaces the named stream with *io.StringIO* object.

使用輸出串流的範例:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

使用輸入串流的範例:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**disable_faulthandler**()

A context manager that temporary disables *faulthandler*.

test.support.**gc_collect**()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.**disable_gc**()

A context manager that disables the garbage collector on entry. On exit, the garbage collector is restored to its prior state.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

Context manager to swap out an attribute with a new object.

用法:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**swap_item**(*obj*, *attr*, *new_val*)

Context manager to swap out an item with a new object.

用法:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**flush_std_streams**()

Call the `flush()` method on *sys.stdout* and then on *sys.stderr*. It can be used to make sure that the logs order is consistent before writing into stderr.

在 3.11 版被加入.

test.support.**print_warning**(*msg*)

Print a warning into *sys.__stderr__*. Format the message as: `f"Warning -- {msg}"`. If *msg* is made of multiple lines, add `"Warning -- "` prefix to each line.

在 3.9 版被加入.

test.support.**wait_process**(*pid*, *, *exitcode*, *timeout=None*)

Wait until process *pid* completes and check that the process exit code is *exitcode*.

如果行程退出代號不等於 *exitcode* 則引發*AssertionError*。

If the process runs longer than *timeout* seconds (*SHORT_TIMEOUT* by default), kill the process and raise an *AssertionError*. The timeout feature is not available on Windows.

在 3.9 版被加入.

test.support.**calcobjsize**(*fmt*)

Return the size of the `PyObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.**calcvobjsize**(*fmt*)

Return the size of the `PyVarObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.**checksizeof**(*test*, *o*, *size*)

For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

@test.support.**anticipate_failure**(*condition*)

A decorator to conditionally mark tests with *unittest.expectedFailure()*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

test.support.**system_must_validate_cert**(*f*)

A decorator that skips the decorated test on TLS certification validation failures.

@test.support.**run_with_locale**(*catstr*, **locales*)

A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

@test.support.**run_with_tz**(*tz*)

A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.**requires_freebsd_version**(**min_version*)

Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, the test is skipped.

@test.support.**requires_linux_version**(**min_version*)

Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, the test is skipped.

@test.support.**requires_mac_version**(**min_version*)

Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, the test is skipped.

@test.support.**requires_gil_enabled**

Decorator for skipping tests on the free-threaded build. If the *GIL* is disabled, the test is skipped.

@test.support.**requires_IEEE_754**

Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.**requires_zlib**

如果*zlib* 不存在則跳過測試的裝飾器。

@test.support.**requires_gzip**

如果*gzip* 不存在則跳過測試的裝飾器。

@test.support.**requires_bz2**

如果*bz2* 不存在則跳過測試的裝飾器。

@test.support.**requires_lzma**

如果*lzma* 不存在則跳過測試的裝飾器。

@test.support.**requires_resource**(*resource*)

如果 *resource* 不可用則跳過測試的裝飾器。

@test.support.**requires_docstrings**

Decorator for only running the test if *HAVE_DOCSTRINGS*.

@test.support.**requires_limited_api**

Decorator for only running the test if Limited C API is available.

@test.support.**cpython_only**

Decorator for tests only applicable to CPython.

@test.support.**impl_detail**(*msg=None*, ***guards*)

Decorator for invoking *check_impl_detail()* on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

@test.support.**thread_unsafe**(*reason=None*)

Decorator for marking tests as thread-unsafe. This test always runs in one thread even when invoked with `--parallel-threads`.

@test.support.**no_tracing**

Decorator to temporarily turn off tracing for the duration of the test.

@test.support.**refcount_test**

Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)

大記憶體測試的裝飾器。

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

@test.support.**bigaddrspacetest**

Decorator for tests that fill the address space.

test.support.**linked_to_musl**()

Return `False` if there is no evidence the interpreter was compiled with `musl`, otherwise return a version triple, either `(0, 0, 0)` if the version is unknown, or the actual version if it is known. Intended for use in `skip` decorators. `emscripten` and `wasi` are assumed to be compiled with `musl`; otherwise `platform.libc_ver` is checked.

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)

Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the exception. If *offset* is not `None`, compares to the offset of the exception.

test.support.**open_urlresource**(*url*, **args*, ***kw*)

Open *url*. If open fails, raises *TestFailed*.

test.support.**reap_children**()

Use this at the end of `test_main` whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)

Get an attribute, raising *unittest.SkipTest* if *AttributeError* is raised.

test.support.**catch_unraisable_exception**()

Context manager catching unraisable exception using *sys.unraisablehook()*.

Storing the exception value (`cm.unraisable.exc_value`) creates a reference cycle. The reference cycle is broken explicitly when the context manager exits.

Storing the object (`cm.unraisable.object`) can resurrect it if it is set to an object which is being finalized. Exiting the context manager clears the stored object.

用法：

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...
```

(繼續下一頁)

(繼續上一頁)

```
    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

在 3.8 版被加入.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the *unittest* `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

這預設會跳過以'_' 開頭的私有屬性，但會包含所有魔術方法，即以'__' 開頭和結尾的方法。

在 3.5 版被加入.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)

Run *code* in subinterpreter. Raise *unittest.SkipTest* if *tracemalloc* is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)

Assert instances of *cls* are deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *not_exported=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper *__module__* attribute. If provided, it will be added to the automatically detected ones.

The *not_exported* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

用法範例:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

在 3.6 版被加入.

test.support.**skip_if_broken_multiprocessing_synchronize**()

Skip tests if the `multiprocessing.synchronize` module is missing, if there is no available semaphore implementation, or if creating a lock raises an *OSError*.

在 3.10 版被加入.

test.support.**check_disallow_instantiation**(*test_case*, *tp*, **args*, ***kwds*)

Assert that type *tp* cannot be instantiated using *args* and *kwds*.

在 3.10 版被加入.

test.support.**adjust_int_max_str_digits**(*max_digits*)

This function returns a context manager that will change the global *sys.set_int_max_str_digits()* setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

在 3.11 版被加入.

test.support 模組定義了以下類🄵:

class test.support.**SuppressCrashReport**

A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

在兩個平台上，舊值會被 `__exit__()` 還原。

class test.support.**SaveSignals**

Class to save and restore signal handlers registered by the Python signal handler.

save(*self*)

Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)

Set the signal numbers from the *save()* dictionary to the saved handler.

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)

Try to match a single stored value (*dv*) with a supplied value (*v*).

26.10 test.support.socket_helper --- 用於 socket 測試的工具

The *test.support.socket_helper* module provides support for socket tests.

在 3.9 版被加入.

test.support.socket_helper.**IPV6_ENABLED**

Set to `True` if IPv6 is enabled on this host, `False` otherwise.

test.support.socket_helper.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is *AF_INET*, *SOCK_STREAM*), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or *bind_port()* should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer *bind_port()* over *find_unused_port()* where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.socket_helper.**bind_port**(*sock*, *host=HOST*)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is *AF_INET* and `sock.type` is *SOCK_STREAM*, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.socket_helper.**bind_unix_socket**(*sock*, *addr*)

Bind a Unix socket, raising *unittest.SkipTest* if *PermissionError* is raised.

@test.support.socket_helper.**skip_unless_bind_unix_socket**

A decorator for running tests that require a functional `bind()` for Unix sockets.

test.support.socket_helper.**transient_internet**(*resource_name*, *, *timeout=30.0*, *errnos=()*)

A context manager that raises *ResourceDenied* when various issues with the internet connection manifest themselves as exceptions.

26.11 test.support.script_helper --- 用於 Python 執行測試的工具

test.support.script_helper 模組提供 Python 的🄴本執行測試的支援。

test.support.script_helper.**interpreter_requires_environment**()

Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting `PYTHONHOME` is one way to get most of the testsuite to run in that situation. `PYTHONPATH` or `PYTHONUSERSITE` are other common environment variables that might impact whether or not the interpreter can start.

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)

Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

在 3.9 版的變更: 此函式不再從 *stderr* 中移除空白。

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the *__cleanenv* keyword-only parameter is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the *__isolated* keyword-only parameter is set to `False`.

在 3.9 版的變更: 此函式不再從 *stderr* 中移除空白。

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

更多選項請見*assert_python_ok()*。

在 3.9 版的變更: 此函式不再從 *stderr* 中移除空白。

test.support.script_helper.**spawn_python**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)

Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

test.support.script_helper.**kill_python**(*p*)

Run the given *subprocess.Popen* process until completion and return stdout.

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is `False`, append `.py` to the name. Return the full script path.

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

Create zip file at *zip_dir* and *zip_basename* with extension `zip` which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing `(full path, full path of archive name)`.

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)

Create a directory named *pkg_dir* containing an `__init__` file with *init_source* as its contents.

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

26.12 test.support.bytecode_helper --- 用於測試位元組碼能正確🄵生的支援工具

test.support.bytecode_helper 模組提供測試和檢查位元組碼🄵生的支援。

在 3.9 版被加入.

此模組定義了以下類🄵:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)

This class has custom assertion methods for inspecting bytecode.

BytecodeTestCase.**get_disassembly_as_string**(*co*)

Return the disassembly of *co* as string.

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Return instr if *opname* is found, otherwise throws *AssertionError*.

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Throws *AssertionError* if *opname* is found.

26.13 test.support.threading_helper --- Utilities for threading tests

The *test.support.threading_helper* module provides support for threading tests.

在 3.10 版被加入.

test.support.threading_helper.**join_thread**(*thread*, *timeout=None*)

Join a *thread* within *timeout*. Raise an *AssertionError* if thread is still alive after *timeout* seconds.

@test.support.threading_helper.**reap_threads**

Decorator to ensure the threads are cleaned up even if the test fails.

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)

Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised; an example would be *threading.Event.set()*. start_threads will attempt to join the started threads upon exit.

test.support.threading_helper.**threading_cleanup**(**original_values*)

Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.threading_helper.**threading_setup**()

Return current thread count and copy of dangling threads.

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)

Context manager to wait until all threads created in the with statement exit.

test.support.threading_helper.**catch_threading_exception**()

Context manager catching *threading.Thread* exception using *threading.excepthook()*.

當捕捉到例外時會設定的屬性:

- exc_type
- exc_value
- exc_traceback
- thread

參🄵*threading.excepthook()* 文件。

這些屬性會在離開情境管理器時被🄵除。

用法:

```
with threading_helper.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...
```

(繼續下一頁)

(繼續上一頁)

```
    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

在 3.8 版被加入.

test.support.threading_helper.**run_concurrently**(*worker_func*, *nthreads*, *args=()*, *kwargs={}*)

Run the worker function concurrently in multiple threads. Re-raises an exception if any thread raises one, after all threads have finished.

26.14 test.support.os_helper --- 用於 os 測試的工具

test.support.os_helper 模組提供 os 測試的支援。

在 3.10 版被加入.

test.support.os_helper.**FS_NONASCII**

A non-ASCII character encodable by *os.fsencode()*.

test.support.os_helper.**SAVEDCWD**

設定🄵*os.getcwd()*。

test.support.os_helper.**TESTFN**

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.os_helper.**TESTFN_NONASCII**

Set to a filename containing the *FS_NONASCII* character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can't work.

test.support.os_helper.**TESTFN_UNENCODABLE**

Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNDECODABLE**

Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNICODE**

Set to a non-ASCII name for a temporary file.

class test.support.os_helper.**EnvironmentVarGuard**

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

在 3.1 版的變更: 新增字典介面。

class test.support.os_helper.**FakePath**(*path*)

Simple *path-like object*. It implements the *__fspath__()* method which just returns the *path* argument. If *path* is an exception, it will be raised in `__fspath__()`.

EnvironmentVarGuard.**set**(*envvar*, *value*)

Temporarily set the environment variable `envvar` to the value of `value`.

EnvironmentVarGuard.**unset**(*envvar*, **others*)

暫時取消一個或多個環境變數。

在 3.14 版的變更: 可以取消多於一個環境變數。

test.support.os_helper.**can_symlink**()

如果作業系統支援符號連結則回傳 `True`，否則回傳 `False`。

test.support.os_helper.**can_xattr**()

如果作業系統支援 xattr 則回傳 `True`，否則回傳 `False`。

test.support.os_helper.**change_cwd**(*path*, *quiet=False*)

A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.os_helper.**create_empty_file**(*filename*)

Create an empty file with *filename*. If it already exists, truncate it.

test.support.os_helper.**fd_count**()

Count the number of open file descriptors.

test.support.os_helper.**fs_is_case_insensitive**(*directory*)

Return `True` if the file system for *directory* is case-insensitive.

test.support.os_helper.**make_bad_fd**()

Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.os_helper.**rmdir**(*filename*)

Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file, which is needed due to antivirus programs that can hold files open and prevent deletion.

test.support.os_helper.**rmtree**(*path*)

Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. As with *rmdir()*, on Windows platforms this is wrapped with a wait loop that checks for the existence of the files.

@test.support.os_helper.**skip_unless_symlink**

A decorator for running tests that require support for symbolic links.

@test.support.os_helper.**skip_unless_xattr**

A decorator for running tests that require support for xattr.

test.support.os_helper.**temp_cwd**(*name='tempcwd'*, *quiet=False*)

A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is `None`, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is `False` and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

test.support.os_helper.**temp_dir**(*path=None*, *quiet=False*)

A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is `None`, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.os_helper.**temp_umask**(*umask*)

A context manager that temporarily sets the process umask.

test.support.os_helper.**unlink**(*filename*)

Call *os.unlink()* on *filename*. As with *rmdir()*, on Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

26.15 test.support.import_helper --- 用於 import 測試的工具

test.support.import_helper 模組提供 import 測試的支援。

在 3.10 版被加入.

test.support.import_helper.**forget**(*module_name*)

Remove the module named *module_name* from `sys.modules` and delete any byte-compiled files of the module.

test.support.import_helper.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

blocked is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`.

如果無法引入指定的模組則此函式會引發*ImportError*。

用法範例:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

在 3.1 版被加入.

test.support.import_helper.**import_module**(*name*, *deprecated=False*, ***, *required_on=()*)

This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against *sys.platform*.

在 3.1 版被加入.

test.support.import_helper.**modules_setup**()

回傳*sys.modules* 的🄵本。

test.support.import_helper.**modules_cleanup**(*oldmodules*)

Remove modules except for *oldmodules* and `encodings` in order to preserve internal cache.

test.support.import_helper.**unload**(*name*)

從 `sys.modules` 中🄵除 *name*。

test.support.import_helper.**make_legacy_pyc**(*source*)

Move a **PEP 3147/PEP 488** pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

class test.support.import_helper.**CleanImport**(**module_names*)

A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a *DeprecationWarning* on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.import_helper.**DirsOnSysPath**(**paths*)

A context manager to temporarily add directories to *sys.path*.

This makes a copy of *sys.path*, appends any directories given as positional arguments, then reverts *sys.path* to the copied settings when the context ends.

Note that *all* *sys.path* modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

26.16 test.support.warnings_helper --- 用於 warnings 測試的工具

test.support.warnings_helper 模組提供 warnings 測試的支援。

在 3.10 版被加入.

test.support.warnings_helper.**ignore_warnings**(***, *category*)

Suppress warnings that are instances of *category*, which must be *Warning* or a subclass. Roughly equivalent to *warnings.catch_warnings()* with *warnings.simplefilter('ignore', category=category)*. For example:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # do something
```

在 3.8 版被加入.

test.support.warnings_helper.**check_no_resource_warning**(*testcase*)

Context manager to check that no *ResourceWarning* was raised. You must remove the object which may emit *ResourceWarning* before the end of the context manager.

test.support.warnings_helper.**check_syntax_warning**(*testcase*, *statement*, *errtext=''*, ***, *lineno=1*, *offset=None*)

Test for syntax warning in *statement* by attempting to compile *statement*. Test also that the *SyntaxWarning* is emitted only once, and that it will be converted to a *SyntaxError* when turned into error. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the emitted *SyntaxWarning* and raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the warning and exception. If *offset* is not `None`, compares to the offset of the exception.

在 3.8 版被加入.

test.support.warnings_helper.**check_warnings**(**filters*, *quiet=True*)

A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with *warnings.simplefilter()* set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised

by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

如果🄵有指定引數，預設🄵:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *`catch_warnings()`* is available via the recorder object's *`warnings`* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, *`check_warnings()`* would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

在 3.2 版的變更: 新的可選引數 *filters* 和 *quiet*。

class test.support.warnings_helper.**WarningsRecorder**

Class used to record warnings for unit tests. See documentation of *`check_warnings()`* above for more details.

CHAPTER 27

除錯與效能分析

這些函式庫幫助你進行 Python 程式開發：除錯器允許你在程式碼中單步 (step) 執行、分析堆🄵框 (stack frames) 以及設置中斷點 (breakpoints) 等，效能分析工具執行程式碼🄵提供關於執行時間的詳細分析，讓你找到程式中的瓶頸 (bottlenecks)。事件稽核 (auditing events) 提供執行時期行🄵的可見性，否則的話可能需要更侵入性的除錯或修補。

27.1 稽核事件表

這張表包含了所有在 CPython 運行環境 (runtime) 與標準函式庫對於*sys.audit()* 或 PySys_Audit() 的呼叫所觸發的事件。這些呼叫是在 3.8 或更新的版本中被新增（請見 **PEP 578**）。

請參考*sys.addaudithook()* 及 PySys_AddAuditHook() 來了解如何處理這些事件。

這張表是從 CPython 文件🄵生的，可能不包含其它實作所觸發的事件。請參考你的運行環境 (runtime) 特定文件來了解實際會觸發的事件。

Audit event	Arguments
_thread.start_new_thread	function, args, kwargs
array.__new__	typecode, initializer
builtins.breakpoint	breakpointhook
builtins.id	id
builtins.input	prompt
builtins.input/result	result
code.__new__	code, filename, name, argcount, posonlyargcount, kwonlyargcount, nlocals, st
compile	source, filename
cpython.PyConfig_Set	name, value
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.remote_debugger_script	script_path
cpython.run_command	command
cpython.run_file	filename
cpython.run_interactivehook	hook
cpython.run_module	module-name
cpython.run_startup	filename
cpython.run_stdin	

表格 1 – 繼續上一頁

Audit event	Arguments
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.memoryview_at	address, size, readonly
ctypes.set_errno	errno
ctypes.set_exception	code
ctypes.set_last_error	error
ctypes.string_at	ptr, size
ctypes.wstring_at	ptr, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	

表格 1 – 繼續上一頁

Audit event	Arguments
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdrives	
os.listmounts	volume
os.listvolumes	
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path, dir_fd
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol

表格 1 – 繼續上一頁

Audit event	Arguments
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name
sqlite3.connect	database
sqlite3.connect/handle	connection_handle
sqlite3.enable_load_extension	connection, enabled
sqlite3.load_extension	connection, path
subprocess.Popen	executable, args, cwd, env
sys._current_exceptions	
sys._current_frames	
sys._getframe	frame
sys._getframemodulename	depth
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.monitoring.register_callback	func
sys.remote_exec	pid
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility
syslog.setlogmask	maskpri
syslog.syslog	priority, message
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
time.sleep	secs
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access
winreg.OpenKey/result	key
winreg.PyHKEY.Detach	key
winreg.QueryInfoKey	key
winreg.QueryReflectionKey	key
winreg.QueryValue	key, sub_key, value_name
winreg.SaveKey	key, file_name
winreg.SetValue	key, sub_key, type, value

下列事件是🄴部觸發的，與任何 CPython 的公開 API 🄵無關🄵：

稽核事件	引數
_winapi.CreateFile	`file_name`, `desired_access`, `share_mode`, `creation_disposition`, `flags_and_attributes`
_winapi.CreateJunction	`src_path`, `dst_path`
_winapi.CreateNamedPi	`name`, `open_mode`, `pipe_mode`
_winapi.CreatePipe	
_winapi.CreateProcess	`application_name`, `command_line`, `current_directory`
_winapi.OpenProcess	`process_id`, `desired_access`
_winapi.TerminateProce	`handle`, `exit_code`
_posixsubpro-cess.fork_exec	`exec_list`, `args`, `env`
ctypes.PyObj_FromPtr	`obj`

在 3.14 版被加入: 🄵部稽核事件 `_posixsubprocess.fork_exec`。

27.2 bdb --- 偵錯器框架

原始碼： Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

有定義以下例外：

exception bdb.**BdbQuit**

由*Bdb* 類🄴所引發的例外，用來退出偵錯器。

bdb 模組也定義了兩個類🄵：

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called *bpbynumber* and by `(file, line)` pairs through *bplist*. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated *file name* should be in canonical form. If a *funcname* is defined, a breakpoint *hit* will be counted when the first line of that function is executed. A *conditional* breakpoint always counts a *hit*.

Breakpoint 實例有以下方法：

deleteMe()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()

Mark the breakpoint as enabled.

disable()

Mark the breakpoint as disabled.

bpformat()

Return a string with all the information about the breakpoint, nicely formatted:

- Breakpoint number.
- Temporary status (del or keep).

- File/line position.
- Break condition.
- Number of times to ignore.
- Number of times hit.

在 3.2 版被加入.

bpprint(*out=None*)

Print the output of *bpformat()* to the file *out*, or if it is `None`, to standard output.

Breakpoint 實例有以下屬性:

file

Breakpoint 的檔案名稱。

line

Line number of the *Breakpoint* within *file*.

temporary

`True` if a *Breakpoint* at (file, line) is temporary.

cond

Condition for evaluating a *Breakpoint* at (file, line).

funcname

Function name that defines whether a *Breakpoint* is hit upon entering the function.

enabled

如*Breakpoint* 有被🄵用則🄵 `True`。

bpbynumber

Numeric index for a single instance of a *Breakpoint*.

bplist

Dictionary of *Breakpoint* instances indexed by (*file*, *line*) tuples.

ignore

Number of times to ignore a *Breakpoint*.

hits

Count of the number of times a *Breakpoint* has been hit.

class bdb.**Bdb**(*skip=None*, *backend='settrace'*)

The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

The *backend* argument specifies the backend to use for *Bdb*. It can be either `'settrace'` or `'monitoring'`. `'settrace'` uses *sys.settrace()* which has the best backward compatibility. The `'monitoring'` backend uses the new *sys.monitoring* that was introduced in Python 3.12, which can be much more efficient because it can disable unused events. We are trying to keep the exact interfaces for both backends, but there are some differences. The debugger developers are encouraged to use the `'monitoring'` backend to achieve better performance.

在 3.1 版的變更: 新增 *skip* 引數。

在 3.14 版的變更: 新增 *backend* 參數。

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)

Return canonical form of *filename*.

For real file names, the canonical form is an operating-system-dependent, *case-normalized absolute path*. A *filename* with angle brackets, such as `"<stdin>"` generated in interactive mode, is returned unchanged.

start_trace(*self*)

Start tracing. For `'settrace'` backend, this method is equivalent to `sys.settrace(self.trace_dispatch)`

在 3.14 版被加入.

stop_trace(*self*)

Stop tracing. For `'settrace'` backend, this method is equivalent to `sys.settrace(None)`

在 3.14 版被加入.

reset()

Set the `botframe`, `stopframe`, `returnframe` and *quitting* attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.
- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)

If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)

If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)

If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)

If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the *quitting* flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

is_skipped_line(*module_name*)

Return `True` if *module_name* matches any skip pattern.

stop_here(*frame*)

Return `True` if *frame* is below the starting frame in the stack.

break_here(*frame*)

Return `True` if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from *effective()*.

break_anywhere(*frame*)

Return `True` if any breakpoint exists for *frame*'s filename.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)

Called from *dispatch_call()* if a break might stop inside the called function.

argument_list is not used anymore and will always be `None`. The argument is kept for backwards compatibility.

user_line(*frame*)

Called from *dispatch_line()* when either *stop_here()* or *break_here()* returns `True`.

user_return(*frame*, *return_value*)

Called from *dispatch_return()* when *stop_here()* returns `True`.

user_exception(*frame*, *exc_info*)

Called from *dispatch_exception()* when *stop_here()* returns `True`.

do_clear(*arg*)

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()

Stop after one line of code.

set_next(*frame*)

Stop on the next line in or below the given frame.

set_return(*frame*)

Stop when returning from the given frame.

set_until(*frame*, *lineno=None*)

Stop when the line with the *lineno* greater than the current one is reached or when returning from current frame.

set_trace([*frame*])

Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

在 3.13 版的變更: *set_trace()* will enter the debugger immediately, rather than on the next line of code to be executed.

set_continue()

Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()

Set the `quitting` attribute to `True`. This raises *BdbQuit* in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the *canonic()* method.

clear_break(*filename*, *lineno*)

Delete the breakpoints in *filename* and *lineno*. If none were set, return an error message.

clear_bpbynumber(*arg*)

Delete the breakpoint which has the index *arg* in the *Breakpoint.bpbynumber*. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)

Delete all breakpoints in *filename*. If none were set, return an error message.

clear_all_breaks()

Delete all existing breakpoints. If none were set, return an error message.

get_bpbynumber(*arg*)

Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a *ValueError* is raised.

在 3.2 版被加入.

get_break(*filename*, *lineno*)

Return `True` if there is a breakpoint for *lineno* in *filename*.

get_breaks(*filename*, *lineno*)

Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)

Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()

Return all breakpoints that are set.

Derived classes and clients can call the following methods to disable and restart events to achieve better performance. These methods only work when using the `'monitoring'` backend.

disable_current_event()

Disable the current event until the next time *restart_events()* is called. This is helpful when the debugger is not interested in the current line.

在 3.14 版被加入.

restart_events()

Restart all the disabled events. This function is automatically called in `dispatch_*` methods after `user_*` methods are called. If the `dispatch_*` methods are not overridden, the disabled events will be restarted after each user interaction.

在 3.14 版被加入.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)

Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix=': '*)

Return a string with information about a stack entry, which is a `(frame, lineno)` tuple. The return string contains:

- The canonical filename which contains the frame.
- 函式名稱或 `"<lambda>"`。
- 輸入引數。
- 回傳值。
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*, *globals=None*, *locals=None*)

Debug a statement executed via the *exec()* function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)

Debug an expression executed via the *eval()* function. *globals* and *locals* have the same meaning as in *run()*.

runctx(*cmd*, *globals*, *locals*)

For backwards compatibility. Calls the *run()* method.

runcall(*func*, */*, **args*, ***kwds*)

Debug a single function call, and return its result.

最後，這個模組定義了以下函式：

bdb.**checkfuncname**(*b*, *frame*)

Return `True` if we should break here, depending on the way the *Breakpoint* *b* was set.

If it was set via line number, it checks if *b.line* is the same as the one in *frame*. If the breakpoint was set via *function name*, we have to check we are in the right *frame* (the right function) and if we are on its first executable line.

bdb.**effective**(*file*, *line*, *frame*)

Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in *bplist* for the (*file*, *line*) (which must exist) that is *enabled*, for which *checkfuncname()* is true, and that has neither a false *condition* nor positive *ignore* count. The *flag*, meaning that a temporary breakpoint should be deleted, is `False` only when the *cond* cannot be evaluated (in which case, *ignore* count is ignored).

If no such entry exists, then `(None, None)` is returned.

bdb.**set_trace**()

Start debugging with a *Bdb* instance from caller's frame.

27.3 faulthandler --- 傾印 Python 回溯

在 3.3 版被加入.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call *faulthandler.enable()* to install fault handlers for the *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS*, and *SIGILL* signals. You can also enable them at startup by setting the PYTHONFAULTHANDLER environment variable or by using the -X faulthandler command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the sigaltstack() function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The backslashreplace error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed: the most recent call is shown first.

By default, the Python traceback is written to *sys.stderr*. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to *faulthandler.enable()*.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

The *Python Development Mode* calls *faulthandler.enable()* at Python startup.

也參考

***pdb* 模組**
Interactive source code debugger for Python programs.

***traceback* 模組**
Standard interface to extract, format and print stack traces of Python programs.

27.3.1 Dumping the traceback

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)

Dump the tracebacks of all threads into *file*. If *all_threads* is False, dump only the current thread.

也參考

traceback.print_tb(), which can be used to print a traceback object.

在 3.5 版的變更: Added support for passing file descriptor to this function.

27.3.2 Dumping the C stack

在 3.14 版被加入.

faulthandler.**dump_c_stack**(*file=sys.stderr*)

Dump the C stack trace of the current thread into *file*.

If the Python build does not support it or the operating system does not provide a stack trace, then this prints an error in place of a dumped C stack.

C Stack Compatibility

If the system does not support the C-level `backtrace(3)` or `dladdr1(3)`, then C stack dumps will not work. An error will be printed instead of the stack.

Additionally, some compilers do not support *CPython's* implementation of C stack dumps. As a result, a different error may be printed instead of the stack, even if the operating system supports dumping stacks.

備🄵

Dumping C stacks can be arbitrarily slow, depending on the DWARF level of the binaries in the call stack.

27.3.3 Fault handler state

`faulthandler.`**`enable`**(*file=sys.stderr*, *all_threads=True*, *c_stack=True*)

Enable the fault handler: install handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` and `SIGILL` signals to dump the Python traceback. If *all_threads* is `True`, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled: see *issue with file descriptors*.

If *c_stack* is `True`, then the C stack trace is printed after the Python traceback, unless the system does not support it. See `dump_c_stack()` for more information on compatibility.

在 3.5 版的變更: Added support for passing file descriptor to this function.

在 3.6 版的變更: On Windows, a handler for Windows exception is also installed.

在 3.10 版的變更: The dump now mentions if a garbage collector collection is running if *all_threads* is true.

在 3.14 版的變更: Only the current thread is dumped if the *GIL* is disabled to prevent the risk of data races.

在 3.14 版的變更: The dump now displays the C stack trace if *c_stack* is true.

`faulthandler.`**`disable`**()

Disable the fault handler: uninstall the signal handlers installed by `enable()`.

`faulthandler.`**`is_enabled`**()

Check if the fault handler is enabled.

27.3.4 Dumping the tracebacks after a timeout

`faulthandler.`**`dump_traceback_later`**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or `cancel_dump_traceback_later()` is called: see *issue with file descriptors*.

This function is implemented using a watchdog thread.

在 3.5 版的變更: Added support for passing file descriptor to this function.

在 3.7 版的變更: This function is now always available.

`faulthandler.`**`cancel_dump_traceback_later`**()

Cancel the last call to `dump_traceback_later()`.

27.3.5 Dumping the traceback on a user signal

`faulthandler.`**`register`**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

Register a user signal: install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by *`unregister()`*: see *issue with file descriptors*.

Not available on Windows.

在 3.5 版的變更: Added support for passing file descriptor to this function.

`faulthandler.`**`unregister`**(*signum*)

Unregister a user signal: uninstall the handler of the *signum* signal installed by *`register()`*. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

27.3.6 Issue with file descriptors

`enable()`, *`dump_traceback_later()`* and *`register()`* keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if *`os.dup2()`* is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

27.3.7 范例

Example of a segmentation fault on Linux with and without enabling the fault handler:

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/opt/python/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>

Current thread's C stack trace (most recent call first):
  Binary file "/opt/python/python", at _Py_DumpStack+0x42 [0x5b27f7d7147e]
  Binary file "/opt/python/python", at +0x32dcbd [0x5b27f7d85cbd]
  Binary file "/opt/python/python", at +0x32df8a [0x5b27f7d85f8a]
  Binary file "/usr/lib/libc.so.6", at +0x3def0 [0x77b73226bef0]
  Binary file "/usr/lib/libc.so.6", at +0x17ef9c [0x77b7323acf9c]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-linux-gnu.
↪so", at +0xcdf6 [0x77b7315dddf6]
  Binary file "/usr/lib/libffi.so.8", at +0x7976 [0x77b73158f976]
  Binary file "/usr/lib/libffi.so.8", at +0x413c [0x77b73158c13c]
  Binary file "/usr/lib/libffi.so.8", at ffi_call+0x12e [0x77b73158ef0e]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-linux-gnu.
↪so", at +0x15a33 [0x77b7315e6a33]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-linux-gnu.
↪so", at +0x164fa [0x77b7315e74fa]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-linux-gnu.
↪so", at +0xc624 [0x77b7315dd624]
  Binary file "/opt/python/python", at _PyObject_MakeTpCall+0xce [0x5b27f7b73883]
  Binary file "/opt/python/python", at +0x11bab6 [0x5b27f7b73ab6]
  Binary file "/opt/python/python", at PyObject_Vectorcall+0x23 [0x5b27f7b73b04]
  Binary file "/opt/python/python", at _PyEval_EvalFrameDefault+0x490c [0x5b27f7cbb302]
  Binary file "/opt/python/python", at +0x2818e6 [0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab [0x5b27f7cd9aab]
```

(繼續下一頁)

(繼續上一頁)

```
  Binary file "/opt/python/python", at PyEval_EvalCode+0xc5 [0x5b27f7cd9ba3]
  Binary file "/opt/python/python", at +0x255957 [0x5b27f7cad957]
  Binary file "/opt/python/python", at +0x255ab4 [0x5b27f7cadab4]
  Binary file "/opt/python/python", at _PyEval_EvalFrameDefault+0x6c3e [0x5b27f7cbd634]
  Binary file "/opt/python/python", at +0x2818e6 [0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab [0x5b27f7cd9aab]
  Binary file "/opt/python/python", at +0x11b6e1 [0x5b27f7b736e1]
  Binary file "/opt/python/python", at +0x11d348 [0x5b27f7b75348]
  Binary file "/opt/python/python", at +0x11d626 [0x5b27f7b75626]
  Binary file "/opt/python/python", at PyObject_Call+0x20 [0x5b27f7b7565e]
  Binary file "/opt/python/python", at +0x32a67a [0x5b27f7d8267a]
  Binary file "/opt/python/python", at +0x32a7f8 [0x5b27f7d827f8]
  Binary file "/opt/python/python", at +0x32ac1b [0x5b27f7d82c1b]
  Binary file "/opt/python/python", at Py_RunMain+0x31 [0x5b27f7d82ebe]
  <truncated rest of calls>
Segmentation fault
```

27.4 pdb --- Python 偵錯器

原始碼： Lib/pdb.py

---

pdb 模組定義了一個 Python 程式的互動式原始碼偵錯器。它支援在原始碼列層級 (source line level) 設定（條件式的）斷點 (breakpoint) 和單步執行、檢視 stack frame（堆🄵框）、列出原始碼、以及在任何 stack frame 情境 (context) 中🄵任意 Python 程式碼求值 (evaluation)。它還支援事後偵錯 (post-mortem debugging)，🄵可以在程式控制下呼叫。

偵錯器是可擴充的——偵錯器實際被定義🄵*Pdb* 類🄵。該類🄵目前🄵有文件，但可以很容易地透過🄵讀原始碼來理解它。擴充套件介面使用了*bdb* 和*cmd* 模組。

也參考

***faulthandler* 模組**
: 用於在出現故障時、超時 (timeout) 後或於接收到使用者訊號時，顯式地轉儲 (dump) Python 回溯 (traceback)。

***traceback* 模組**
: 用於提取、格式化和印出 Python 程式 stack trace（堆🄵追🄵）的標準介面。

自一個執行中程式切入偵錯器的典型用法🄵插入：

```
import pdb; pdb.set_trace()
```

或：

```
breakpoint()
```

到你想切入偵錯器的位置，然後執行程式，就可以單步執行上述陳述式之後的程式碼，🄵可以使用*continue* 命令來離開偵錯器、繼續執行。

在 3.7 版的變更: 當使用預設值呼叫時，可以使用🄵建的*breakpoint()* 來取代 `import pdb; pdb.set_trace()`。

```
def double(x):
   breakpoint()
   return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

偵錯器的提示字元是 (Pdb)，這表示你處於偵錯模式：

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

在 3.3 版的變更: 透過 *readline* 模組達成的 tab 補全可用於補全本模組的命令和命令的引數，例如會提供目前的全域和區域名稱以作🄵 p 命令的引數。 你還可以從命令列叫用 *pdb* 來偵錯其他🄵本。例如：

```
python -m pdb [-c command] (-m module | -p pid | pyfile) [args ...]
```

當作🄵模組叫用時，如果被偵錯的程序不正常地退出，pdb 將自動進入事後偵錯。事後偵錯後（或程式正常退出後），pdb 將重新🄵動程式。自動重新🄵動會保留 pdb 的狀態（例如斷點），🄵且在大多數情🄵下比在程式退出時退出偵錯器更有用。

-c, --command <command>

就像在 .pdbrc 檔案中給定的那樣來執行命令；請參🄵偵錯器命令。

在 3.2 版的變更: 新增了 -c 選項。

-m <module>

以類似於 python -m 的方式來執行模組。與🄵本一樣，偵錯器將在模組的第一列之前暫停執行。

在 3.7 版的變更: 新增了 -m 選項。

-p, --pid <pid>

Attach to the process with the specified PID.

在 3.14 版被加入.

To attach to a running Python process for remote debugging, use the -p or --pid option with the target process's PID:

```
python -m pdb -p 1234
```

備🄵

Attaching to a process that is blocked in a system call or waiting for I/O will only work once the next bytecode instruction is executed or when the process receives a signal.

在偵錯器控制下執行陳述式的典型用法是：

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

檢查一個損壞程式的典型用法：

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
```

(繼續下一頁)

(繼續上一頁)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

在 3.13 版的變更: **PEP 667** 的實作意味著透過 pdb 進行的名稱賦予 (name assignments) 會立即影響有效作用域，即使在*最佳化作用域*🄵運作也是如此。

本模組定義了下列函式，每個函式進入偵錯器的方式略有不同：

pdb.**run**(*statement*, *globals=None*, *locals=None*)

在偵錯器控制下執行 *statement*（以字串或程式碼物件形式給定）。偵錯提示字元會在執行任何程式碼前出現；你可以設定斷點🄵輸入*continue*，或也可以使用*step* 或*next* 逐步執行陳述式（這些命令在下面都有🄵明）。可選引數 *globals* 和 *locals* 指定程式碼執行的環境；預設使用*__main__* 模組的字典。（請參🄵🄵建函式*exec()* 或*eval()* 的🄵明。）

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)

在偵錯器控制下🄴 *expression* 求值（以字串或程式碼物件形式給定）。當*runeval()* 回傳時，它回傳 *expression* 的值。除此之外，該函式與*run()* 類似。

pdb.**runcall**(*function*, **args*, ***kwds*)

使用給定的引數呼叫 *function*（只可以是函式或方法物件，不能是字串）。*runcall()* 回傳的是所呼叫函式的回傳值。偵錯器提示字元將在進入函式後立即出現。

pdb.**set_trace**(*, *header=None*, *commands=None*)

在呼叫此函式的 stack frame 進入偵錯器。用於在程式中給定之處寫死 (hard-code) 一個斷點，即便該程式碼不在偵錯狀態（如斷言失敗時）。如有給定 *header*，它將在偵錯正要開始前被印出到控制台。如果給定了 *commands* 引數，它會是在偵錯器🄴動時要執行的命令清單。

在 3.7 版的變更: 僅限關鍵字引數 *header*。

在 3.13 版的變更: *set_trace()* 將立即進入偵錯器，而不是在下一列要執行的程式碼中。

在 3.14 版被加入: *commands* 引數。

awaitable pdb.**set_trace_async**(*, *header=None*, *commands=None*)

async version of *set_trace()*. This function should be used inside an async function with `await`.

```
async def f():
    await pdb.set_trace_async()
```

`await` statements are supported if the debugger is invoked by this function.

在 3.14 版被加入.

pdb.**post_mortem**(*t=None*)

進入所給定例外或回溯物件的事後偵錯。如果🄵有給定，預設使用目前正在處理的例外，或如果🄵有例外則會引發 `ValueError`。

在 3.13 版的變更: 新增對例外物件的支援。

pdb.**pm**()

進入在*sys.last_exc* 中發現的例外的事後偵錯。

pdb.**set_default_backend**(*backend*)

There are two supported backends for pdb: `'settrace'` and `'monitoring'`. See *bdb.Bdb* for details. The user can set the default backend to use if none is specified when instantiating *Pdb*. If no backend is specified, the default is `'settrace'`.

> **備🄵**
>
> `breakpoint()` and `set_trace()` will not be affected by this function. They always use `'monitoring'` backend.

在 3.14 版被加入.

`pdb.`**`get_default_backend`**`()`

Returns the default backend for pdb.

在 3.14 版被加入.

`run*` 函式和`set_trace()` 都是🄵名，用於實例化 (instantiate) `Pdb` 類🄵🄵呼叫同名方法。如果要使用更多功能，則必須自己執行以下操作：

`class` `pdb.`**`Pdb`**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*, *mode=None*, *backend=None*, *colorize=False*)

`Pdb` 是偵錯器類🄵。

completekey、*stdin* 與 *stdout* 引數會被傳到底層的`cmd.Cmd` 類🄵；請於該文件🄵讀相關🄵述。

如果給定 *skip* 引數，則它必須是一個給出 glob 樣式之模組名稱的🄵代器。如果遇到匹配這些樣式的模組，偵錯器將不會進入來自該模組的 frame。[1]

預設情🄵下，當你發出`continue` 命令時，Pdb 會🄵 SIGINT 訊號（即使用者在控制台上按下 `Ctrl-C` 時會發送的訊號）設定一個處理程式 (handler)，這允許你透過按下 `Ctrl-C` 再次切入偵錯器。如果你希望 Pdb 不影響到 SIGINT 處理程式，請將 *nosigint* 設定🄵 true。

readrc 引數預設🄵 true，它控制 Pdb 是否從檔案系統載入.pdbrc 檔案。

The *mode* argument specifies how the debugger was invoked. It impacts the workings of some debugger commands. Valid values are `'inline'` (used by the breakpoint() builtin), `'cli'` (used by the command line invocation) or `None` (for backwards compatible behaviour, as before the *mode* argument was added).

The *backend* argument specifies the backend to use for the debugger. If `None` is passed, the default backend will be used. See `set_default_backend()`. Otherwise the supported backends are `'settrace'` and `'monitoring'`.

The *colorize* argument, if set to `True`, will enable colorized output in the debugger, if color is supported. This will highlight source code displayed in pdb.

🄵用追🄵 (tracing) 且帶有 *skip* 引數的呼叫示範：

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

不帶引數地引發一個稽核事件 *(auditing event)* `pdb.Pdb`。

在 3.1 版的變更: 新增了 *skip* 參數。

在 3.2 版的變更: 新增了 *nosigint* 參數。以前 SIGINT 處理程式從未被 Pdb 設定過。

在 3.6 版的變更: *readrc* 引數。

在 3.14 版被加入: 新增了 *mode* 引數。

在 3.14 版被加入: 新增了 *backend* 引數。

在 3.14 版被加入: 新增了 *colorize* 引數。

在 3.14 版的變更: Inline breakpoints like `breakpoint()` or `pdb.set_trace()` will always stop the program at calling frame, ignoring the *skip* pattern (if any).

`run`(*statement*, *globals=None*, *locals=None*)

`runeval`(*expression*, *globals=None*, *locals=None*)

`runcall`(*function*, **args*, ***kwds*)

[1] 一個 frame 是否會被認🄵源自特定模組是由該 frame 全域性變數中的 `__name__` 來🄵定的。

set_trace()

請見上面關於這些函式的文件🄵明。

27.4.1 偵錯器命令

下方列出的是偵錯器能認得的命令。如下所示，大多數命令可以縮寫🄵一個或兩個字母。如 `h(elp)` 表示可以輸入 `h` 或 `help` 來輸入幫助命令（但不能輸入 `he` 或 `hel`，也不能是 `H` 或 `Help` 或 `HELP`）。命令的引數必須用空格（空格符 (spaces) 或🄵表符 (tabs)）分隔。在命令語法中，可選引數被括在方括號 (`[]`) 中；使用時請勿輸入方括號。命令語法中的選擇項由🄵🄵 (`|`) 分隔。

輸入一個空白列 (blank line) 將重🄵上次輸入的命令。例外：如果上一個命令是*list* 命令，則會列出接下來的 11 列。

偵錯器無法識🄵的命令將被認🄵是 Python 陳述式，🄵以正在偵錯的程式之情境來執行。Python 陳述式也可以用驚嘆號 (`!`) 作🄵前綴，這是檢視正在偵錯之程式的🄵大方法，甚至可以修改變數或呼叫函式。當此類陳述式發生例外，將印出例外名稱，但偵錯器的狀態不會改變。

在 3.13 版的變更: 現在可以正確辨識🄵執行前綴🄵 pdb 命令的運算式/陳述式。

偵錯器有支援設定🄵*名*。🄵名可以有參數，使得偵錯器對被檢查的情境有一定程度的適應性。

在一列中可以輸入多個以 `;;` 分隔的命令。（不能使用單個 `;`，因🄴它用於分隔傳遞給 Python 剖析器一列中的多個命令。）切分命令🄵運用什🄵高深的方式；輸入總是在第一處 `;;` 被切分開，即使它位於引號🄵的字串之中。對於具有雙分號字串的一個變通解法，是使用隱式字串連接 `';''';'` 或 `";"";"`。

要設定臨時全域變數，請使用*便利變數 (convenience variable)*。便利變數是名稱以 `$` 開頭的變數。例如 `$foo = 1` 會設定一個全域變數 `$foo`，你可以在偵錯器會話 (debugger session) 中使用它。當程式恢復執行時，便利變數將被清除，因此與使用 `foo = 1` 等普通變數相比，它不太會去干擾你的程式。

共有四個預先設定的便利變數：

- `$_frame`：目前正在偵錯的 frame
- `$_retval`：frame 回傳時的回傳值
- `$_exception`：frame 引發例外時的例外
- `$_asynctask`: the asyncio task if pdb stops in an async function

在 3.12 版被加入: 新增了便利變數功能。

在 3.14 版被加入: 新增了 `$_asynctask` 便利變數。

如果 `.pdbrc` 檔案存在於使用者的家目🄵或目前目🄵中，則會使用 `'utf-8'` 編碼讀取🄵執行該檔案，就像在偵錯器提示字元下鍵入該檔案一樣，除了空列和以 `#` 開頭的列會被忽略之外。這對於🄵名設定特🄵有用。如果兩個檔案都存在，則先讀取家目🄵中的檔案，且定義於其中的🄵名可以被本地檔案覆蓋。

在 3.2 版的變更: `.pdbrc` 現在可以包含繼續偵錯的命令，如*continue* 或*next*。以前檔案中的這些命令是無效的。

在 3.11 版的變更: `.pdbrc` 現在使用 `'utf-8'` 編碼讀取。以前它是使用系統區域設定編碼讀取的。

h(elp) [command]

如不帶引數，印出可用的命令列表。引數🄵 *command* 時，印出有關該命令的幫助訊息，`help pdb` 會顯示完整文件（即*pdb* 模組的🄵明字串 (docstring)）。由於 *command* 引數必須是一個識🄵字 (identifier)，若要取得 `!` 命令的幫助訊息則必須輸入 `help exec`。

w(here) [count]

印出 stack trace，最新的 frame 會位於底部。如果 *count* 🄵 0，則印出目前的 frame 條目。如果 *count* 🄵負數，印出最舊的 *count* 個 frame。如果 *count* 🄵正數，印出最新的 *count* 個 frame。箭頭（`>`）表示目前的 frame，它也🄵定了大多數命令的情境。

在 3.14 版的變更: *count* argument is added.

d(own) [count]

在 stack trace 中，將目前 frame 向下移動 *count* 級（預設🄵 1 級，移往較新的 frame）。

u(p) [count]

在 stack trace 中，將目前 frame 向上移動 *count* 級（預設🄵 1 級，移向較舊的 frame）。

b(reak) [([filename:]lineno | function) [, condition]]

如帶有 *lineno* 引數，則在目前檔案中的 *lineno* 列處設定中斷。列號可以以 *filename* 和冒號🄵前綴，以指定另一個檔案（可能是尚未載入的檔案）中的斷點。該檔案會在*sys.path* 上搜尋。可接受的 *filename* 形式🄵 /abspath/to/file.py、relpath/file.py、module 和 package.module。

如帶有 *function* 引數，在該函式🄵的第一個可執行陳述式處設定中斷。*function* 可以是任何其求值結果🄵目前命名空間中函式的運算式。

如果第二個引數存在，它是一個運算式，在斷點生效前其必須求值🄵 true

如果不帶引數執行會列出所有斷點資訊，包括每個斷點、命中該斷點的次數、目前的忽略次數以及關聯的條件（如存在）。

每個斷點都有賦予一個編號，所有其他斷點命令都參照該編號。

tbreak [([filename:]lineno | function) [, condition]]

臨時斷點，在第一次遇見時會自動被🄵除。它的引數與*break* 相同。

cl(ear) [filename:lineno | bpnumber ...]

如帶有 *filename:lineno* 引數，則清除此列上的所有斷點。如果引數是空格分隔的斷點編號列表，則清除這些斷點。如果不帶引數則清除所有斷點（但會先提示確認）。

disable bpnumber [bpnumber ...]

停用斷點，斷點以空格分隔的斷點編號列表來給定。停用斷點表示它不會導致程式停止執行，但是與清除斷點不同，停用的斷點將保留在斷點列表中🄵且可以（重新）🄵用。

enable bpnumber [bpnumber ...]

🄵用指定的斷點。

ignore bpnumber [count]

🄵給定的斷點編號設定忽略計數。如果省略 *count* 則忽略計數設定🄵 0。當忽略計數🄵 0 時，斷點將變🄵有效狀態。當非 0 時，每次到達斷點，且斷點🄵有被禁用，且任何關聯的條件被求值🄵 true，則 *count* 就會遞🄵。

condition bpnumber [condition]

🄵斷點設定一個新 *condition*，🄵一個運算式，且其求值結果🄵 true 時斷點才會起作用。如果🄵有給定 *condition*，則🄵除任何現有條件，也就是不🄵斷點設定條件。

commands [bpnumber]

🄵編號是 *bpnumber* 的斷點指定一系列命令。命令🄵容出現在後續的幾列中。輸入僅包含 end 的一列來結束命令列表。例如：

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

要🄵除斷點上的所有命令，請輸入 commands 🄵立即以 end 結尾，也就是不指定任何命令。

不帶有 *bpnumber* 引數則 commands 會關聯到上一個設定的斷點。

可以使用斷點命令來重新🄵動程式，只需使用*continue* 或*step* 命令，或其他可以繼續執行程式的命令。

如果指定了某個繼續執行程式的命令（目前包括*continue*、*step*、*next*、*return*、*until*、*jump*、*quit* 及它們的縮寫）將終止命令列表（就像該命令後馬上跟著 end）。因🄵在任何時候繼續執行下去（即使是簡單的 next 或 step），都可能會遇到另一個斷點，該斷點可能具有自己的命令列表，這會導致無法確定要執行哪個列表。

If the list of commands contains the silent command, or a command that resumes execution, then the breakpoint message containing information about the frame is not displayed.

在 3.14 版的變更: Frame information will not be displayed if a command that resumes execution is present in the command list.

s(tep)
執行目前列，在第一個可以停止的位置（在被呼叫的函式🄵部或在目前函式的下一列）停止。

n(ext)
繼續執行，直至執行到目前函式的下一列或者目前函式回傳🄵止。（*next* 和*step* 之間的區🄵在於*step* 會在被呼叫函式的🄵部停止、而*next*（幾乎）全速執行被呼叫的函式，🄵僅在目前函式的下一列停止。）

unt(il) `[lineno]`
如果不帶引數則繼續執行，直到列號比目前的列大時停止。

如帶有 *lineno* 則繼續執行，直到到達列號大於或等於 *lineno* 的那一列。在這兩種情🄵下，目前 frame 回傳時也會停止。

在 3.2 版的變更: 允許明確給定一個列號。

r(eturn)
繼續執行，直到目前的函式回傳。

c(ont(inue))
繼續執行，除非遇到斷點才停下來。

j(ump) `lineno`
設定即將執行的下一列，僅可用於堆🄵中最底部的 frame。這讓你可以跳回去🄵再次執行程式碼，或者往前跳以跳過不想執行的程式碼。

需要注意的是，不是所有的跳轉都是被允許的 -- 例如不能跳轉到 `for` 🄵圈的中間或跳出 `finally` 子句。

l(ist) `[first[, last]]`
列出目前檔案的原始碼。如果不帶引數，則列出目前列周圍的 11 列，或繼續先前的列表操作。如果用 . 作🄵引數，則列出目前列周圍的 11 列。如果帶有一個引數，則列出那一列周圍的 11 列。如果帶有兩個引數，則列出給定範圍中的程式碼；如果第二個引數小於第一個引數，則將其直譯🄵要列出的列數。

目前 frame 中的目前列會用 -> 標記出來。如果正在偵錯一個例外，且引發或傳遞該例外的那一列不是目前列，則會用 >> 來標記該列。

在 3.2 版的變更: 新增了 >> 標記。

ll | `longlist`
列出目前函式或 frame 的所有原始碼。相關列的標記方式與*list* 相同。

在 3.2 版被加入.

a(rgs)
印出目前函式的引數及它們目前的值。

p `expression`
在目前情境中🄵 *expression* 求值🄵印出其值。

> **備🄵**
>
> 也可以使用 `print()`，但它不是一個偵錯器命令 --- 它會執行 Python *print()* 函式。

pp `expression`
與*p* 命令類似，除了 *expression* 的值是使用*pprint* 模組美化後印出來的。

whatis expression
印出 *expression* 的型🄵。

source expression
嘗試取得 *expression* 的原始碼🄵顯示它。

在 3.2 版被加入.

display [expression]
每次在目前 frame 中停止執行時，顯示 *expression* 的值（如果有變更）。

如果不帶有 *expression*，則列出目前 frame 的所有運算式。

> **備🄵**
>
> display 會對 *expression* 求值🄵將結果與之前 *expression* 的求值結果進行比較，因此當結果可變時，display 可能無法取得其變更。

範例如下：

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

display 不會意識到 `lst` 已更改，因🄵其求值結果在比較之前已被 `lst.append(1)` 原地 (in place) 修改：

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

你可以運用🄵🄵機制的一些技巧來使其能🄵運作：

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

在 3.2 版被加入.

undisplay [expression]
不再顯示目前 frame 中的 *expression*。如果不帶有 *expression*，則清除目前 frame 的所有顯示運算式。

在 3.2 版被加入.

interact

在從目前作用域的區域和全域命名空間初始化的新全域命名空間中🄵動互動式直譯器（使用*code* 模組）。可使用 `exit()` 或 `quit()` 退出直譯器🄵回到偵錯器。

> **備🄵**
>
> 由於 `interact` 🄵程式碼執行建立了一個新的專用命名空間，因此變數的賦值不會影響原始命名空間，但是對任何被參照的可變物件的修改將像往常一樣反映在原始命名空間中。

在 3.2 版被加入.

在 3.13 版的變更: `exit()` 和 `quit()` 可用來退出*interact* 命令。

在 3.13 版的變更: *interact* 將其輸出導向到偵錯器的輸出通道 (output channel)，而不是*sys.stderr*。

alias `[name [command]]`

建立一個名🄵 *name* 的🄵名，用於執行 *command*。*command* 不得用引號包起來。可被替🄵的參數要用 `%1`、`%2`、... 和 `%9` 標示，而 `%*` 則被所有參數替🄵。如果省略 *command*，則顯示 *name* 的目前🄵名。如果未給定引數，則列出所有🄵名。

巢狀🄵名是允許的，且可包含能在 pdb 提示字元下合法輸入的任何🄵容。請注意，🄵部 pdb 命令可以被🄵名所覆蓋。這樣的命令在🄵名被移除前都將被隱藏。🄵名會遞🄵地應用到命令列的第一個單詞；該列🄵的其他單詞則不會受影響。

作🄵範例，這🄵列出了兩個有用的🄵名（特🄵是放在 `.pdbrc` 檔案中時）：

```
# 印出實例變數（用法如 "pi classInst"）
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} = {%1.__dict__[k]}")
# 印出 self 中的實例變數
alias ps pi self
```

unalias `name`

🄵除指定的🄵名 *name*。

! `statement`

在目前 stack frame 的情境中執行（單列）*statement*。除非陳述式的第一個單詞類似於偵錯器命令，否則可以省略驚嘆號，例如：

```
(Pdb) ! n=42
(Pdb)
```

要設定全域變數，你可以在同一列的賦值命令前面加上 `global` 陳述式，例如：

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run `[args ...]`

restart `[args ...]`

重新🄵動已偵錯完畢的 Python 程式。如果提供了 *args*，它將以*shlex* 分割，🄵將結果用作新的*sys.argv*。歷史記🄵、斷點、操作和偵錯器選項均會被保留。*restart* 是*run* 的🄵名。

在 3.14 版的變更: *run* and *restart* commands are disabled when the debugger is invoked in `'inline'` mode.

q(uit)

離開偵錯器，執行中的程式會被中止。輸入檔案結尾 (end-of-file) 相當於*quit*。

A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. Either `y`, `Y`, `<Enter>` or `EOF` will confirm the quit.

在 3.14 版的變更: A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. After the confirmation, the debugger will call *`sys.exit()`* immediately, instead of raising *`bdb.BdbQuit`* in the next trace event.

debug `code`

進入一個遞🄴偵錯器，逐步執行 *code*（這是要在目前環境中執行的任意運算式或陳述式）。

retval

印出目前函式最後一次回傳的回傳值。

exceptions `[excnumber]`

列出鏈接例外 (chained exceptions)，或在其間跳轉。

當使用 `pdb.pm()` 或 `Pdb.post_mortem(...)` 於鏈接例外而不是回溯時，它允許使用者在鏈接例外之間移動，使用 `exceptions` 命令以列出例外，🄴使用 `exceptions <number>` 切🄴到該例外。

範例如下：

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
    1 / x

 out()
```

呼叫 `pdb.pm()` 將允許在例外之間移動：

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

在 3.13 版被加入.

27.5 Python 的分析器

原始碼： Lib/profile.py 與 Lib/pstats.py

27.5.1 分析器簡介

cProfile and *profile* provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the *pstats* module.

The Python standard library provides two different implementations of the same profiling interface:

1. *cProfile* is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. *profile*, a pure Python module whose interface is imitated by *cProfile*, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

備🄵

The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is *timeit* for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

27.5.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use *profile* instead of *cProfile* if the latter is not available on your system.)

The above action would run *re.compile()* and print profile results like the following:

```
      214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 __init__.py:250(compile)
     1    0.000    0.000    0.001    0.001 __init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000 _compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000 _parser.py:937(parse)
     1    0.000    0.000    0.000    0.000 _compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000 _parser.py:435(_parse_sub)
```

The first line indicates that 214 calls were monitored. Of those calls, 207 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: cumulative time` indicates the output is sorted by the `cumtime` values. The column headings include:

ncalls
: for the number of calls.

tottime
: for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
: is the quotient of `tottime` divided by `ncalls`

cumtime
: is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall
: is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)
: provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The *pstats.Stats* class reads profile results from a file and formats them in various ways.

The files *cProfile* and *profile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

-o <output_file>
: Writes the profile results to a file instead of to stdout.

-s <sort_order>
: Specifies one of the *sort_stats()* sort values to sort the output by. This only applies when *-o* is not supplied.

-m <module>
: Specifies that a module is being profiled instead of a script.

  在 3.7 版被加入: 新增 `-m` 選項到*cProfile*。

  在 3.8 版被加入: 新增 `-m` 選項到*profile*。

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

27.5.3 `profile` 與 `cProfile` 模組參考文件

Both the *profile* and *cProfile* modules provide the following functions:

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals mappings for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

`class` `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The *Profile* class can also be used as a context manager (supported only in *cProfile* module. see 情境管理器型🄵):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

    pr.print_stats()
```

在 3.8 版的變更: 新增情境管理器的支援。

enable()

Start collecting profiling data. Only in *cProfile*.

disable()

Stop collecting profiling data. Only in *cProfile*.

create_stats()

Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)

Create a *Stats* object based on the current profile and print the results to stdout.

The *sort* parameter specifies the sorting order of the displayed statistics. It accepts a single key or a tuple of keys to enable multi-level sorting, as in *Stats.sort_stats*.

在 3.13 版被加入: *print_stats()* now accepts a tuple of keys.

dump_stats(*filename*)

Write the results of the current profile to *filename*.

run(*cmd*)

Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)

Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, */*, **args*, ***kwargs*)

Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

27.5.4 `Stats` 類🄵

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)

This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats 物件有以下方法：

strip_dirs()

This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)

This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)

Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)

This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	Valid enum Arg	含義
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name（檔案名稱）
`'filename'`	SortKey.FILENAME	file name（檔案名稱）
`'module'`	N/A	file name（檔案名稱）
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	列號
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

在 3.7 版被加入: Added the SortKey enum.

reverse_order()

This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)

This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will be interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)

This method for the *Stats* class prints a list of all functions that called each function in the profiled

database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)

This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

get_stats_profile()

This method returns an instance of StatsProfile, which contains a mapping of function names to instances of FunctionProfile. Each FunctionProfile instance holds information related to the function's profile such as how long the function took to run, how many times it was called, etc...

在 3.9 版被加入: Added the following dataclasses: StatsProfile, FunctionProfile. Added the following function: get_stats_profile.

27.5.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required in order to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

27.5.6 限制

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with *profile* than with the lower-overhead *cProfile*. For this reason, *profile* provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average)

removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

27.5.7 校正

The profiler of the *profile* module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see 限制).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

27.5.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile

`your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see 校正). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating-point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile

your_time_func should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if your_integer_time_func returns times measured in thousands of seconds, you would construct the Profile instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the cProfile.Profile class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal _lsprof module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

27.6 timeit --- 測量小量程式片段的執行時間

原始碼： Lib/timeit.py

該模組提供了一種對少量 Python 程式碼進行計時的簡單方法。它有一個命令列介面和一個可呼叫介面，它避免了許多測量執行時間的常見陷阱。另請參🄵由 O'Reilly 出版的 *Python 錦囊妙計 (Python Cookbook)* 第二版中 Tim Peters 所寫的「演算法」章節的介紹。

27.6.1 基礎范例

以下範例展示了如何使用命令列介面來比較三個不同的運算式：

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

這可以透過*Python 介面*來實現：

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

也可以在*Python 介面*傳遞可呼叫物件：

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

但請注意，僅當使用命令列介面時*timeit()* 才會自動確定重🄵次數。在範例章節中有更進階的範例。

27.6.2 Python 介面

該模組定義了三個便利函式和一個公開類🄵：

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

使用給定的陳述式、*setup* 程式碼和 *timer* 函式建立一個*Timer* 實例，🄵執行其*timeit()* 方法 *number* 次。可選的 *globals* 引數指定會在其中執行程式碼的命名空間。

在 3.5 版的變更: 新增 *globals* 選用參數。

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

使用給定的陳述式、*setup* 程式碼和 *timer* 函式建立一個 *Timer* 實例，🄵使用給定的 *repeat* 計數和 *number* 來運行其 *repeat()* 方法。可選的 *globals* 引數指定會在其中執行程式碼的命名空間。

在 3.5 版的變更: 新增 *globals* 選用參數。

在 3.7 版的變更: *repeat* 的預設值從 3 更改🄵 5。

timeit.**default_timer**()

預設計時器始終🄵 time.perf_counter()，會回傳浮點秒數。另一種方法是 time.perf_counter_ns，會回傳整數奈秒。

在 3.3 版的變更: *time.perf_counter()* 現在是預設計時器。

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

用於計時小程式碼片段執行速度的類🄵。

建構函式接受一個要計時的陳述式、一個用於設定的附加陳述式和一個計時器函式。兩個陳述式都預設🄵 `'pass'`；計時器函式會與平台相依（請參🄵模組文件字串 (doc string)）。*stmt* 和 *setup* 還可以包含由 `;` 或🄵行符號分隔的多個陳述式，只要它們不包含多行字串文字即可。預設情🄵下，該陳述式將在 timeit 的命名空間🄵執行；可以透過將命名空間傳遞給 *globals* 來控制此行🄵。

要測量第一個陳述式的執行時間，請使用 *timeit()* 方法。*repeat()* 和 *autorange()* 方法是多次呼叫 *timeit()* 的便捷方法。

setup 的執行時間不包含在總體運行計時中。

stmt 和 *setup* 參數還可以接受無需引數即可呼叫的物件。這會把對它們的呼叫嵌入到計時器函式中，然後由 *timeit()* 去執行。請注意，在這種情🄵下，因有額外的函式呼叫，時間開銷 (timing overhead) 會稍大一些。

在 3.5 版的變更: 新增 *globals* 選用參數。

timeit(*number=1000000*)

主陳述式執行 *number* 次的時間。這將執行一次設定陳述式，然後回傳多次執行主陳述式所需的時間。預設計時器以浮點形式回傳秒數，引數是🄵圈的次數，預設🄵一百萬次。要使用的主陳述式、設定陳述式和計時器函式會被傳遞給建構函式。

> **備🄵**
>
> 預設情🄵下 *timeit()* 在計時期間會暫時關閉垃圾回收。這種方法的優點是它使獨立時序更具可比較性，缺點是 GC 可能是被測函式性能的重要組成部分。如果是這樣，可以將 GC 作🄵 *setup* 字串中的第一個陳述式以重新🄵用。例如:
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

自動🄵定呼叫 *timeit()* 次數。

這是一個便捷函式，它重🄵呼叫 *timeit()* 以使得總時間 >= 0.2 秒，再回傳最終結果（🄵圈數、該🄵圈數所花費的時間）。它以 1、2、5、10、20、50... 的順序遞增數字來呼叫 *timeit()* 直到所用時間至少🄵 0.2 秒。

如果有給定 *callback* 且不是 `None`，則每次試驗後都會使用兩個引數來呼叫它: `callback(number, time_taken)`。

在 3.6 版被加入.

repeat(*repeat=5*, *number=1000000*)

呼叫 *timeit()* 數次。

這是一個方便的函式，它會重🄵呼叫 *timeit()* 🄵回傳結果列表。第一個引數指定呼叫 *timeit()* 的次數，第二個引數指定 *timeit()* 的 *number* 引數。

> **備🄵**
>
> 人們很容易根據結果向量來計算出平均值和標准差🄵將其作🄴依歸，然而這🄵不是很有用。在典型情🄵下，最低值給出了機器運行給定程式碼片段的速度的下限；結果向量中較高的值通常不是由 Python 速度的變化所引起，而是由干擾計時精度的其他行程造成的。因此，結果中的*min()* 可能是你應該感興趣的唯一數字。在解讀該數據後，你應該查看整個向量🄴以常識判讀而非單純仰賴統計資訊。

在 3.7 版的變更: *repeat* 的預設值從 3 更改🄴 5。

print_exc(*file=None*)

從計時程式碼印出回溯 (traceback) 的輔助函式。

典型用法：

```
t = Timer(...)       # 在 try/except 之外
try:
    t.timeit(...)    # 或 t.repeat(...)
except Exception:
    t.print_exc()
```

相對於標准回溯的優點是，已編譯模板中的原始程式碼會被顯示出來。可選的 *file* 引數指定回溯的發送位置；它預設🄵*sys.stderr*。

27.6.3 命令列介面

當從命令列作🄵程式呼叫時，請使用以下形式：

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h] [statement ...]
```

其中之以下選項：

-n N, **--number**=N

執行'statement' 多少次

-r N, **--repeat**=N

計時器重🄵多少次（預設 5）

-s S, **--setup**=S

會在一開始執行一次的陳述式（預設🄵 `pass`）

-p, **--process**

若要測量行程時間 (process time) 而非🄴鐘時間 (wallclock time)，請使用*time.process_time()* 而不是預設的*time.perf_counter()*

在 3.3 版被加入.

-u, **--unit**=U

指定定時器輸出的時間單位；可以選擇 `nsec`、`usec`、`msec` 或 `sec`

在 3.5 版被加入.

-v, **--verbose**

印出原始計時結果；重🄵執行以獲得更高的數字精度

-h, **--help**

印出一條簡短的使用訊息🄵退出

可以透過將每一列陳述式指定🄵單獨引數來給定多列陳述式；可透過將引數括在引號中🄴使用前導空格以實現縮進列 (indented lines)。多個*-s* 選項的作用類似。

如果🄵有給定*-n*，則透過嘗試從序列 1, 2, 5, 10, 20, 50, ... 中增加數字來計算合適的🄵圈次數，直到總時間至少🄵 0.2 秒。

default_timer() 測量可能會受到同一台機器上運行的其他程式的影響，因此，當需要精確計時時，最好的做法是重🄴計時幾次🄴使用最佳時間。*-r* 選項對此很有用；在大多數情🄴下，預設的重🄵 5 次可能就足🄵了。你可以使用*time.process_time()* 來測量 CPU 時間。

備🄵

執行 pass 陳述式會🄴生一定的基本開銷。這🄴的程式碼🄵不試圖隱藏它，但你應該意識到它的存在。可以透過不帶引數呼叫程式來測量這個基本開銷，且不同 Python 版本之間的基本開銷可能有所不同。

27.6.4 范例

可以提供一個僅會在開始時執行一次的設定陳述式：

```
$ python -m timeit -s "text = 'sample string'; char = 'g'" "char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'" "text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

輸出中包含三個欄位。🄴圈計數，它告訴你每次計時🄵圈🄵重🄴運行陳述式主體的次數。重🄵計數（「最好的 5 次」）告訴你計時🄴圈重🄵了多少次。以及最後陳述式主體在計時🄴圈的最佳的幾次重🄵執行🄵平均花費的時間。也就是🄴，最快的幾次重🄵執行所花費的總時間除以🄴圈計數。

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

同樣可以使用*Timer* 類🄴及其方法來完成：

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

以下範例展示如何對包含多行的運算式進行計時。這🄴我們使用*hasattr()* 與 `try`/`except` 來測試缺失和存在之物件屬性🄴比較其花費 (cost)：

```
$ python -m timeit "try:" "  str.__bool__" "except AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
```

(繼續下一頁)

(繼續上一頁)

```
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要讓*timeit* 模組存取你定義的函式，你可以傳遞一個包含 import 陳述式的 *setup* 參數：

```
def test():
    """一個笨笨的測試函式"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

另一種選擇是將*globals()* 傳遞給 *globals* 參數，這將導致程式碼在目前的全域命名空間中執行，這比單獨指定 import 更方便：

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 `trace` --- 追🄵或追查 Python 陳述式執行

原始碼： Lib/trace.py

---

trace 模組可用來追🄴程式執行過程、🄵生帶🄵解的陳述式涵蓋範圍清單、輸出呼叫者／被呼叫者的關🄵，以及在程式執行期間被執行的函式串列。它可以在其他程式中使用，也可以從命令列🄴動。

也參考

Coverage.py
一款受歡迎的第三方涵蓋範圍工具，提供 HTML 輸出與分支涵蓋等進階功能。

27.7.1 命令列用法

trace 模組可從命令列執行，用法可以非常簡單，如：：

```
python -m trace --count -C . somefile.py ...
```

上述命令會執行 somefile.py，F將F生執行期間所引入的所有 Python 模組的帶F解串列，輸出至目前目F。

--help

顯示用法F結束程式。

--version

顯示模組版本F結束程式。

在 3.8 版被加入: 新增 --module 選項，允許執行可執行模組。

主要選項

執行*trace* 時，至少必須指定以下選項之一。*--listfuncs* 選項與*--trace* 及*--count* 選項互斥。當提供*--listfuncs* 時，將不接受*--count* 或*--trace*，反之亦然。

-c, --count

在程式執行結束時F生一組帶F解串列檔案，顯示每個陳述式被執行的次數。參見下方的*--coverdir*、*--file* 及*--no-report*。

-t, --trace

執行時即時顯示每一行。

-l, --listfuncs

顯示程式執行過程中被呼叫的函式。

-r, --report

從先前使用*--count* 與*--file* 選項執行過的程式生成帶F解的串列，不會執行任何程式碼。

-T, --trackcalls

顯示程式執行時F生的呼叫關F。

修飾子（Modifier）

-f, --file=<file>

指定用來累積多次追F結果的檔案之名稱。應與*--count* 選項搭配使用。

-C, --coverdir=<dir>

報告檔案的輸出目F。package.module 的涵蓋範圍報告將寫入檔案：*dir/package/module*.cover。

-m, --missing

在F生帶F解的串列時，使用 >>>>>> 標記未被執行的程式碼行。

-s, --summary

使用*--count* 或*--report* 時，會將每個處理過的檔案摘要輸出至 stdout。

-R, --no-report

不F生帶F解的串列。若你打算多次使用*--count* 執行程式，F在最後再統一F生串列，這會很有幫助。

-g, --timing

在每一行前加上自程式F動以來的經過時間。僅在追F時使用。

篩選子（Filter）

這些選項可以重🄵使用多次。

--ignore-module=<mod>

忽略所指定的模組名稱及其子模組（若🄵套件）。引數可以是以逗號分隔的名稱串列。

--ignore-dir=<dir>

忽略指定目🄵及其子目🄵中的所有模組與套件。引數可以是以*os.pathsep* 分隔的目🄵串列。

27.7.2 程式介面

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)

建立一個物件以追🄵單一陳述式或運算式的執行。所有參數皆🄵可選的。*count* 🄵用行數統計，*trace* 🄵用逐行追🄵執行，*countfuncs* 可列出執行期間呼叫的函式，*countcallers* 追🄵呼叫關🄵，*ignoremods* 是要忽略的模組或套件名稱串列，*ignoredirs* 是其模組或套件應被忽略的目🄵串列，*infile* 是讀取儲存計數資料的檔案名稱，*outfile* 是要寫入更新後計數資料的檔案名稱，*timing* 則會顯示自開始追🄵以來的時間戳記。

run(*cmd*)

執行命令，🄵依目前追🄵參數收集執行統計資料。*cmd* 必須是可傳入*exec()* 的字串或程式碼物件。

runctx(*cmd*, *globals=None*, *locals=None*)

在指定的全域與區域命名空間中，根據目前的追🄵參數執行命令🄵收集統計資料。若未指定，*globals* 與 *locals* 預設🄵空字典。

runfunc(*func*, */*, **args*, ***kwds*)

在目前的追🄵參數下，使用*Trace* 物件控制呼叫 *func*，🄵傳入指定的引數。

results()

回傳一個*CoverageResults* 物件，包含指定*Trace* 實例中所有先前對 `run`、`runctx` 與 `runfunc` 呼叫的累積結果。不會重設累積的追🄵結果。

class trace.**CoverageResults**

一個用來儲存涵蓋範圍結果的容器，由*Trace.results()* 建立。不應由使用者直接建立。

update(*other*)

合🄵來自另一個*CoverageResults* 物件的資料。

write_results(*show_missing=True*, *summary=False*, *coverdir=None*, ***, *ignore_missing_files=False*)

寫入涵蓋範圍結果。將 *show_missing* 設🄵真可顯示未被執行的程式碼行；將 *summary* 設🄵真則會在輸出中加入每個模組的涵蓋範圍摘要。*coverdir* 指定輸出涵蓋範圍結果檔案的目🄵，若🄵 `None`，則每個原始檔案的結果會輸出至其所在目🄵中。

若 *ignore_missing_files* 🄵 `True`，則會忽略那些已不存在的檔案的涵蓋範圍資料，不會顯示任何錯誤。否則，缺少的檔案將會引發*FileNotFoundError*。

在 3.13 版的變更: 新增 *ignore_missing_files* 參數。

以下是一個簡單範例，展示如何使用程式介面：

```
import sys
import trace

# 建立一個 Trace 物件，指定要忽略的目🄵，
# 🄵設定是否進行追🄵、行數統計或兩者皆做。
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)
```

(繼續下一頁)

(繼續上一頁)

```
# 使用該 tracer 執行新的命令
tracer.run('main()')

# 建立報告，將輸出放在目前目🄵中
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.8 tracemalloc --- 追🄵記憶體配置

在 3.4 版被加入.

原始碼： Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

27.8.1 范例

Display the top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
```

(繼續下一頁)

(繼續上一頁)

```
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *collections* module allocated `244 KiB` to build *namedtuple* types.

更多選項請見*Snapshot.statistics()*。

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), average=139␣
↪B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), average=86␣
↪B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), average=96.0␣
↪KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), average=784␣
↪B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *linecache* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the *importlib* loaded data most recently: on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

更多選項請見*Snapshot.statistics()*。

Record the current and peak size of all traced memory blocks

The following code computes two sums like `0 + 1 + 2 + ...` inefficiently, by creating a list of those numbers. This list consumes a lot of memory temporarily. We can use *get_traced_memory()* and *reset_peak()* to observe the small memory usage after the sum is computed as well as the peak memory usage during the computations:

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

輸出：

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

Using *reset_peak()* ensured we could accurately record the peak during the computation of `small_sum`, even though it is much smaller than the overall peak size of memory blocks since the *start()* call. Without the call to *reset_peak()*, `second_peak` would still be the peak from the computation `large_sum` (that is, equal to `first_peak`). In this case, both peaks are much higher than the final memory usage, and which suggests we could optimise (by removing the unnecessary call to *list*, and writing `sum(range(...))`).

27.8.2 API

函式

`tracemalloc.`**`clear_traces`**`()`

Clear traces of memory blocks allocated by Python.

另請參🄵*stop()*。

`tracemalloc.`**`get_object_traceback`**`(`*obj*`)`

Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

另請參🄵*gc.get_referrers()* 與*sys.getsizeof()* 函式。

`tracemalloc.`**`get_traceback_limit`**`()`

Get the maximum number of frames stored in the traceback of a trace.

The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

tracemalloc.**get_traced_memory**()

Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

tracemalloc.**reset_peak**()

Set the peak size of memory blocks traced by the *tracemalloc* module to the current size.

Do nothing if the *tracemalloc* module is not tracing memory allocations.

This function only modifies the recorded peak size, and does not modify or clear any traces, unlike *clear_traces()*. Snapshots taken with *take_snapshot()* before a call to *reset_peak()* can be meaningfully compared to snapshots taken after the call.

另請參🄵*get_traced_memory()*。

在 3.9 版被加入.

tracemalloc.**get_tracemalloc_memory**()

Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

tracemalloc.**is_tracing**()

`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

另請參🄵*start()* 與*stop()* 函式。

tracemalloc.**start**(*nframe:* int = *1*)

Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

You can still read the original number of total frames that composed the traceback by looking at the *Traceback.total_nframe* attribute.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()

Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()

Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

另請參🄵*get_object_traceback()* 函式。

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive:* bool, *domain:* int)

Filter traces of memory blocks by their address space (domain).

在 3.6 版被加入.

inclusive

If *inclusive* is `True` (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space *domain*.

domain

Address space of a memory block (`int`). Read-only property.

Filter

class tracemalloc.**Filter**(*inclusive:* bool, *filename_pattern:* str, *lineno:* int = *None*, *all_frames:* bool = *False*, *domain:* int = *None*)

Filter on traces of memory blocks.

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

範例:

- `Filter(True, subprocess.__file__)` only includes traces of the *subprocess* module
- `Filter(False, tracemalloc.__file__)` excludes traces of the *tracemalloc* module
- `Filter(False, "<unknown>")` excludes empty tracebacks

在 3.5 版的變更: The `'.pyo'` file extension is no longer replaced with `'.py'`.

在 3.6 版的變更: 新增*domain* 屬性。

domain

Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive

If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno

Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern

Filename pattern of the filter (`str`). Read-only property.

all_frames

If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**

Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename

Filename (str).

lineno

Line number (int).

Snapshot

class tracemalloc.**Snapshot**

Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot:* Snapshot, *key_type:* str, *cumulative:* bool = *False*)

Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)

Write the snapshot into a file.

Use *load()* to reload the snapshot.

filter_traces(*filters*)

Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

在 3.6 版的變更: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)

Load a snapshot from a file.

另請參🄵*dump()*。

statistics(*key_type:* str, *cumulative:* bool = *False*)

Get statistics as a sorted list of *Statistic* instances grouped by *key_type*:

key_type	描述
'filename'	filename
'lineno'	filename and line number
'traceback'	traceback

If *cumulative* is True, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to 'filename' and 'lineno'.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit

Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces

Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**

Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

另請參🄵*StatisticDiff* 類🄵。

count

Number of memory blocks (`int`).

size

Total size of memory blocks in bytes (`int`).

traceback

Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**

Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count

Number of memory blocks in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

count_diff

Difference of number of memory blocks between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

size

Total size of memory blocks in bytes in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

size_diff

Difference of total size of memory blocks in bytes between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

traceback

Traceback where the memory blocks were allocated, *Traceback* instance.

Trace

class tracemalloc.**Trace**

Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

在 3.6 版的變更: 新增*domain* 屬性。

domain

Address space of a memory block (`int`). Read-only property.

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size

Size of the memory block in bytes (`int`).

traceback

Traceback where the memory block was allocated, *`Traceback`* instance.

Traceback

class tracemalloc.**Traceback**

Sequence of *`Frame`* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to *`get_traceback_limit()`* frames. See the *`take_snapshot()`* function. The original number of frames of the traceback is stored in the *`Traceback.total_nframe`* attribute. That allows to know if a traceback has been truncated by the traceback limit.

The *`Trace.traceback`* attribute is an instance of *`Traceback`* instance.

在 3.7 版的變更: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

total_nframe

Total number of frames that composed the traceback before truncation. This attribute can be set to `None` if the information is not available.

在 3.9 版的變更: The *`Traceback.total_nframe`* attribute was added.

format(*limit=None*, *most_recent_first=False*)

Format the traceback as a list of lines. Use the *`linecache`* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *`traceback.format_tb()`* function, except that *`format()`* does not include newlines.

範例:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

輸出:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 28

軟體封裝與發布

這些函式庫可以幫助你發布和安裝 Python 軟體。雖然這些模組設計是🄵了與 Python 套件索引 (Python Package Index) 結合使用，但它們也可以搭配本地索引伺服器，甚至可以在🄵有任何索引伺服器的情🄵下使用。

28.1 `ensurepip` --- `pip` 安裝器的初始建置 (bootstrapping)

在 3.4 版被加入.

原始碼： Lib/ensurepip

`ensurepip` 套件 (package) 🄵既有的 Python 安裝或🄵擬環境提供 `pip` 安裝器初始建置 (bootstrapping) 的支援。這個初始建置的方式應證了事實——`pip` 是有其獨立發布🄵期的專案，且其最新可用穩定的版本，會與 CPython 直譯器 (interpreter) 之維護和功能發布綁定。

大多數情🄵下，Python 的終端使用者不需要直接叫用此模組（因🄵 `pip` 預設應🄵初始建置），但若安裝 Python 時 `pip` 被省略（或建立一🄵擬環境時），又或是 `pip` 被明確解除安裝時，則此模組的初始建置仍有可能用上。

備🄵

此模組不會通過網路存取。所有需要用來初始建置 `pip` 的元件都已包含在套件之🄵。

也參考

installing-index
對於終端使用者安裝 Python 套件的指引

PEP 453: 在 Python 安裝中的 pip 明確初始建置
此模組的最初設計理念與規範。

可用性: not Android, not iOS, not WASI.

此模組在行動平台或 *WebAssembly 平台* 上不支援。

28.1.1 命令列介面

使用直譯器 (interpreter) 的 `-m` 來叫用命令列介面

最簡單可行的呼叫：

```
python -m ensurepip
```

若 `pip` 未安裝，此呼叫會將其安裝；否則甚🄵也不做。可透過傳入 `--upgrade` 參數來確保 `pip` 的安裝版本至少🄵目前 `ensurepip` 中最新可用的版本：

```
python -m ensurepip --upgrade
```

預設上，`pip` 會被安裝至目前🄵擬環境（若已🄵動🄵擬環境）或系統端的套件（若🄵有🄵動的🄵擬環境）。安裝位置可透過兩個額外的命令列選項來控制：

--root <dir>

安裝 `pip` 到給定的根目🄵 (root directory) 的相對路徑，而不是目前🄵擬環境（若存在）的根目🄵，或目前 Python 安裝版所預設的根目🄵。

--user

安裝 `pip` 到使用者端的套件目🄵，而不是全域安裝到目前的 Python 安裝版（此選項不允許在一🄵動的🄵擬環境中使用）。

預設會安裝 `pipX` 和 `pipX.Y` 🄵本（X.Y 代表用來叫用 `ensurepip` 的 Python 之版本）。安裝的🄵本可透過兩個額外的命令列選項來控制：

--altinstall

若要求一個替代安裝版，則不會安裝 `pipX` 🄵本。

--default-pip

若要求安裝「預設 pip」，則會安裝 `pip` 🄵本，及 2 個常規🄵本。

提供兩種指令選項將會導致例外 (exception) 。

28.1.2 模組 API

ensurepip 開放了兩個用於編寫程式的函式：

ensurepip.**version**()

回傳一個字串，用以標明初始建置時，安裝的 pip 的可行版本號。

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)

在目前或指定的環境之中建立 `pip`。

root 指定一個替代的根目🄵，作🄵安裝的相對路徑。若 *root* 🄵 `None`，則安裝使用目前環境的預設安裝位置。

upgrade 指出是否要將一個既有的較早版本的 `pip` 升級至可用的新版。

user 指出是否要使用使用者的安裝方案而不是全域安裝。

預設會安裝 `pipX` 和 `pipX.Y` 🄵本（X.Y 代表 Python 的目前版本）。

如果🄵用了 *altinstall*，則不會安裝 `pipX`。

如果🄵用了 *default_pip*，則會安裝 `pip`，以及 2 個常規🄵本。

同時🄵用 *altinstall* 跟 *default_pip*，將會觸發 *ValueError*。

verbosity 用來控制初始建置操作🄵，對於 *sys.stdout* 的輸出等級。

引發一個附帶引數 `root` 的稽核事件 `ensurepip.bootstrap`。

> **備🄵**
>
> 初始建置的過程對於 `sys.path` 及 `os.environ` 均有副作用。改🄵在一子行程 (subprocess) 呼叫命令列介面可避免這些副作用。

> **備🄵**
>
> 初始建置的過程也許會安裝 `pip` 所需要的額外的模組，但其他軟體不應該假設這些相依 (dependency) 總是預設存在（因🄵這些相依很可能會在未來版本的 `pip` 中被移除）。

28.2 venv --- 建立🄵擬環境

在 3.3 版被加入.

原始碼： Lib/venv/

`venv` 模組支援建立輕量級的「🄵擬環境」，每個🄵擬環境都有獨立安裝在其*site* 目🄵中的一組 Python 套件。🄵擬環境是建立在既有的 Python 安裝上（稱🄵該🄵擬環境的「基底」Python）之上，預設會與基底環境中的套件隔離，使得僅能存取明確安裝在🄵擬環境中的套件。更多資訊請參🄵🄵擬環境 以及*site* 的🄵擬環境文件。

在使用🄵擬環境中時，像 pip 這樣的常用安裝工具會自動將 Python 套件安裝至🄵擬環境中，而不需要額外指定。

🄵擬環境是（主要的特性）：

- 用來包含特定的 Python 直譯器、函式庫與二進位檔案，以支援某個專案（函式庫或應用程式）。預設情🄵下，這些🄵容會與其他🄵擬環境中的軟體以及作業系統中安裝的 Python 直譯器與函式庫隔離。
- 被包含在一個目🄵中，通常命名🄵專案目🄵下的 `.venv` 或 `venv`，也可能是在用來管理多個🄵擬環境的容器目🄵中，例如 `~/.virtualenvs`。
- 不應提交至 Git 等版本控制系統。
- 被視🄵可🄵🄵的資源——應能簡單地🄵除🄵從頭重新建立。請勿將任何專案程式碼放置於🄵擬環境中。
- 不應移動或🄵🄵——若需在其他位置使用，應重新建立相同的🄵擬環境。

更多關於 Python 🄵擬環境的背景資訊請見 **PEP 405**。

> **也參考**
>
> Python 套件指南：建立與使用🄵擬環境

可用性: not Android, not iOS, not WASI.

此模組在行動平台或*WebAssembly* 平台上不支援。

28.2.1 建立🄵擬環境

🄵擬環境是透過執行 `venv` 模組來建立：

```
python -m venv /path/to/new/virtual/environment
```

執行此命令會建立目標目🄵（同時也會建立任何還不存在的父目🄵）🄵在目🄵中放置一個名🄵 `pyvenv.cfg` 的檔案，其中包含一個指向執行該命令的 Python 安裝路徑的 `home` 鍵。它同時會建立一個 `bin`（在 Windows 上🄵 `Scripts`）子目🄵，其中包含一個 Python 二進位檔案的副本/符號連結（根據建立環境時使用的平台或引數而定）。此外，它還會建立一個 `lib/pythonX.Y/site-packages` 子目🄵（在 Windows 上🄵 `Lib\site-packages`）。如果指定的目🄵已存在，則將重新使用該目🄵。

在 3.5 版的變更: 目前建議使用 `venv` 來建立🄵擬環境。

Deprecated since version 3.6, removed in version 3.8: **pyvenv** 是在 Python 3.3 和 3.4 中建立🄵擬環境的推薦工具，但在 Python 3.5 中已被直接執行 `venv` 所取代。

在 Windows 上，執行以下命令以使用 `venv`：

```
PS> python -m venv C:\path\to\new\virtual\environment
```

如果使用 `-h` 選項執行該命令，將會顯示可用的選項：

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            [--without-scm-ignore-files]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

ENV_DIR

A required argument specifying the directory to create the environment in.

--system-site-packages

Give the virtual environment access to the system site-packages directory.

--symlinks

Try to use symlinks rather than copies, when symlinks are not the default for the platform.

--copies

Try to use copies rather than symlinks, even when symlinks are the default for the platform.

--clear

Delete the contents of the environment directory if it already exists, before environment creation.

--upgrade

Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.

--without-pip

Skips installing or upgrading pip in the virtual environment (pip is bootstrapped by default).

--prompt `<PROMPT>`

Provides an alternative prompt prefix for this environment.

--upgrade-deps

Upgrade core dependencies (pip) to the latest version in PyPI.

--without-scm-ignore-files

Skips adding SCM ignore files to the environment directory (Git is supported by default).

在 3.4 版的變更: 預設會安裝 pip，🄵新增了 `--without-pip` 和 `--copies` 選項

在 3.4 版的變更: 在較早的版本中，如果目標目🄵已存在，除非提供了 `--clear` 或 `--upgrade` 選項，否則會引發錯誤。

在 3.9 版的變更: 新增 `--upgrade-deps` 選項以將 pip 和 setuptools 升級至 PyPI 上的最新版本

在 3.12 版的變更: `setuptools` 不再是 venv 的核心相依套件。

在 3.13 版的變更: 新增 `--without-scm-ignore-files` 選項

在 3.13 版的變更: `venv` 現在預設會🄵 Git 建立 `.gitignore` 檔案。

備🄵

雖然在 Windows 上支援符號連結，但🄵不建議使用。特🄵需要注意的是，在檔案總管中按兩下 `python.exe` 會急切地解析符號連結🄵忽略🄵擬環境。

備🄵

在 Microsoft Windows 上，可能需要通過設置使用者的執行策略來🄵用 `Activate.ps1` 🄵本。你可以發出以下 PowerShell 命令來執行此操作:

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

有關更多資訊，請參🄵關於執行策略。

被建立的 `pyvenv.cfg` 檔案還包括了 `include-system-site-packages` 的鍵，如果使用 `venv` 執行時帶有 `--system-site-packages` 選項，則設置🄵 `true`，否則設置🄵 `false`。

除非 `--without-pip` 選項被提供，否則將叫用*ensurepip* 來🄵動 `pip` 到🄵擬環境中。

可以向 `venv` 提供多個路徑，這樣每個提供的路徑都將根據給定的選項建立一個相同的🄵擬環境。

28.2.2 🄵擬環境如何運作

當 Python 直譯器跑在🄵擬環境時，*sys.prefix* 和*sys.exec_prefix* 會指向🄵擬環境的目🄵，而*sys.base_prefix* 和*sys.base_exec_prefix* 會指向建立🄵擬環境的基礎 Python 的目🄵。檢查 `sys.prefix != sys.base_prefix` 就可以確定目前的直譯器是否跑在🄵擬環境中。

🄵擬環境可以透過位於二進位檔案目🄵中的🄵本「🄵用」(在 POSIX 上🄵 `bin`; 在 Windows 上🄵 `Scripts`) 這會將該目🄵加入到你的 `PATH`，當你運行 **`python`** 時就會叫用該環境的直譯器🄵且執行已安裝的🄵本，而不需要使用完整的路徑。🄵動🄵本的方式因平台而🄵（*<venv>* 需要替🄵成包含🄵擬環境的目🄵路徑）

平台	Shell	🄵動🄵擬環境的指令
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

在 3.4 版被加入: **`fish`** 和 **`csh`** 🄵動🄵本。

在 3.8 版被加入: PowerShell 的🄵動🄵本安裝在 POSIX 上支援 PowerShell Core。

你不用特🄵開🄵🄵擬環境，你可以在呼叫 Python 時指定該環境下 Python 直譯器的完整路徑。此外，所有安裝在環境🄵的🄵本都應該都可以在未🄵用🄵擬環境的情🄵下運行。

🄵了實現這一點，安裝在🄵擬環境中的🄵本會有一個”shebang” 列，此列指向該環境的 Python 直譯器 `#!/<path-to-venv>/bin/python`。這代表無論 `PATH` 的值🄵何，該🄵本都會在直譯器上運行。在 Windows 上，如果你安裝了 launcher，則支援”shebang” 列處理。因此，在 Windows 檔案總管（Windows Explorer）中雙擊已安裝的🄵本，應該可以在🄵有🄵用環境或將其加入 `PATH` 的情🄵下正確地運行。

當🄵擬環境被🄵用時，`VIRTUAL_ENV` 環境變數會被設置🄵該環境的路徑。由於不需要明確🄵用🄵擬環境才能使用它。因此，無法依賴 `VIRTUAL_ENV` 來判斷是否正在使用🄵擬環境。

警告

因□安裝在環境中的□本不應該預期該環境已經被□動，所以它們的 shebang 列會包含環境直譯器的□對路徑。因此，在一般情□下，環境本質上是不可□帶的。你應該使用一個簡單的方法來重新建立一個環境（例如：如果你有一個名□ `requirements.txt` 的需求檔案，你可以使用環境的 `pip install -r requirements.txt` 來安裝環境所需的所有套件）。如果出於某種原因，你需要將環境移至新位置，你應該在所需位置重新建立它，□□除舊位置的環境。如果你移動環境是因□移動了其父目□，你應該在新位置重新建立環境。否則，安裝在該環境中的軟體可能無法正常運作。

你可以在 shell 輸入 `deactivate` 來關閉□擬環境。具體的使用方式因平台而□，是□部實作的細節（通常會使用□本或是 shell 函式）

28.2.3 API

上述提到的高階 method（方法）透過簡單的 API 使用，□第三方□擬環境建立者提供可以依據他們需求來建立環境的客□化機制：*EnvBuilder* class。

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

進行實例化時，class *EnvBuilder* 接受下列的關鍵字引數：

- *system_site_packages* -- □一個 boolean （布林值），□表明系統的 Python site-packages 是否可以在環境中可用（預設□ `False` ）。
- *clear* -- □一個 boolean，如果□ true，則在建立環境之前，□除目標目□□所有存在的□容。
- *symlinks* -- □一個 boolean，□表明是否嘗試與 Python 二進位檔案建立符號連結而不是□□該檔案。
- *upgrade* -- □一個 boolean，若□ true，則會在執行 Python 時□現有的環境進行升級。目的是讓 Python 可以升級到位（預設□ `False`）。
- *with_pip* -- □一個 boolean，若□ true，則確保 pip 有安裝至□擬環境之中。當有 `--default-pip` 的選項時，會使用*ensurepip*。
- *prompt* -- □一個 String（字串），該字串會在□擬環境□動時被使用。（預設□ `None`，代表該環境的目□名稱會被使用）倘若出現特殊字串 `"."` ，則目前目□的 basename 會做□提示路徑使用。
- *upgrade_deps* -- 更新基礎 venv 模組至 PyPI 的最新版本
- *scm_ignore_files*——根據可□代物件中指定的來源控制管理（SCM），建立對應的忽略檔。支援情□取□於是否有名□ `create_{scm}_ignore_file` 的方法。預設唯一支援的值是 `"git"`，透過*create_git_ignore_file()* 方法提供支援。

在 3.4 版的變更: 新增 `with_pip` 參數

在 3.6 版的變更: 新增 `prompt` 參數

在 3.9 版的變更: 新增 `upgrade_deps` 參數

在 3.13 版的變更: 新增 `scm_ignore_files` 參數

EnvBuilder 可以被用作□基底類□。

create(*env_dir*)

透過指定將會容納□擬環境的目標目□來建立一個□擬環境（□對路徑或相對路徑到該目□），也就是在該目□中容納□擬環境。`create` method 將會在指定的目□下建立環境，或是觸發適當的例外。

EnvBuilder class 的 `create` method 會闡述可用的 Hooks 以客□化 subclass （子類□）：

```
def create(self, env_dir):
    """
    在目標目🄵中建立一個🄵擬化的 Python 環境。
    env_dir 是要建立🄵擬環境的目標目🄵。
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

每個 methods *ensure_directories()*、*create_configuration()*、*setup_python()*、*setup_scripts()* 及*post_setup()* 都可以被覆寫。

ensure_directories(*env_dir*)

建立還不存在的環境目🄵及必要的子目🄵，🄵回傳一個情境物件（context object）。這個情境物件只是一個屬性 (例如：路徑) 的所有者，可被其他 method 使用。如果*EnvBuilder* 已被建立且帶有 `clear=True` 的引數，該環境目🄵下的🄵容將被清空，以及所有必要的子目🄵將被重新建立。

回傳的情境物件（context object）其型🄵會是*types.SimpleNamespace*，🄵包含以下屬性：

- `env_dir` —— 🄵擬環境的位置。用於🄵用🄵本中的 `__VENV_DIR__`（參見*install_scripts()*）。
- `env_name` —— 🄵擬環境的名稱。用於🄵用🄵本中的 `__VENV_NAME__`（參見*install_scripts()*）。
- `prompt` ——🄵用🄵本所使用的提示字元。用於🄵用🄵本中的 `__VENV_PROMPT__`（參見*install_scripts()*）。
- `executable` ——🄵擬環境所使用的底層 Python 執行檔。在從其他🄵擬環境建立🄵擬環境的情🄵下，也會一🄵考慮該情境。
- `inc_path` ——🄵擬環境的 include 路徑。
- `lib_path` ——🄵擬環境的 purelib 路徑。
- `bin_path` ——🄵擬環境的🄵本路徑。
- `bin_name` —— 相對於🄵擬環境位置的🄵本路徑名稱。用於🄵用🄵本中的 `__VENV_BIN_NAME__`（參見*install_scripts()*）。
- `env_exe` ——🄵擬環境中 Python 直譯器的名稱。用於🄵用🄵本中的 `__VENV_PYTHON__`（參見*install_scripts()*）。
- `env_exec_cmd` ——Python 直譯器名稱，會將檔案系統重新導向後納入考慮。可用於在🄵擬環境中執行 Python。

在 3.11 版的變更: *venv* 使用*sysconfig 安裝配置方案* 來建構所建立目🄵的路徑。

在 3.12 版的變更: `lib_path` 屬性已新增至情境中，且情境物件已加入文件🄵明。

create_configuration(*context*)

在環境中建立 `pyvenv.cfg` 設定檔。

setup_python(*context*)

在環境中建立 Python 執行檔的🄵本或符號連結。在 POSIX 系統上，若使用了特定的 `python3.x` 執行檔，將建立指向該執行檔的 `python` 與 `python3` 符號連結（除非這些名稱的檔案已存在）。

setup_scripts(*context*)

將符合平台的🄵用🄵本安裝到🄵擬環境中。

upgrade_dependencies(*context*)

更新🄴擬環境中的核心 venv 相依套件（目前🄴 pip）。此動作會透過執行🄴擬環境中的 `pip` 執行檔來完成。

在 3.9 版被加入.

在 3.12 版的變更: setuptools 不再是核心 venv 的相依項目。

post_setup(*context*)

此🄵位方法在🄴擬環境中預先安裝套件，或執行其他建立後的動作，可由第三方實作覆寫。

install_scripts(*context*, *path*)

此方法可由子類🄵中的*setup_scripts()* 或*post_setup()* 呼叫，協助將自訂🄵本安裝到🄵擬環境中。

path 是一個目🄵的路徑，該目🄵應包含子目🄵 `common`、`posix` 和 `nt`；每個子目🄵中都包含指向環境中 `bin` 目🄵的🄵本。在執行部分🄵位符文字替🄵後，將會🄵🄵 `common` 及與*os.name* 對應的目🄵的🄵容：

- `__VENV_DIR__` 會被替🄵🄵環境目🄵的🄵對路徑。
- `__VENV_NAME__` 會被替🄵🄵環境的名稱（即該環境目🄵路徑的最後一段）。
- `__VENV_PROMPT__` 會被替🄵🄵提示字元（即以括號包裹的環境名稱，🄴附帶一個空格）。
- `__VENV_BIN_NAME__` 會被替🄵🄵 bin 目🄵的名稱（可能是 `bin` 或 `Scripts`）。
- `__VENV_PYTHON__` 會被替🄵🄵環境中 Python 執行檔的🄵對路徑。

允許這些目🄵存在（例如當更新既有環境時）。

create_git_ignore_file(*context*)

在🄵擬環境中建立一個 `.gitignore` 檔案，讓 Git 原始碼控管工具忽略整個目🄵。

在 3.13 版被加入.

在 3.7.2 版的變更: Windows 現在使用重新導向🄵本來處理 `python[w].exe`，而不再🄵🄵實際的二進位檔。在 3.7.2 中，只有當從原始碼樹中建構時，*setup_python()* 才會執行操作。

在 3.7.3 版的變更: Windows 會在*setup_python()* 中🄵🄵重新導向🄵本，而非在*setup_scripts()* 中進行。這在 3.7.2 中🄵非如此。若使用符號連結，則會連結至原始的可執行檔。

也有一個方便的模組層級的函式：

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, *, *scm_ignore_files=frozenset()*)

使用提供的關鍵字引數建立一個*EnvBuilder*，🄵以 *env_dir* 作🄵引數呼叫其*create()* 方法。

在 3.3 版被加入.

在 3.4 版的變更: 新增 *with_pip* 參數

在 3.6 版的變更: 新增 *prompt* 參數

在 3.9 版的變更: 新增 *upgrade_deps* 參數

在 3.13 版的變更: 新增 *scm_ignore_files* 參數

28.2.4 一個擴展 EnvBuilder 的范例

以下的🄵本展示了如何透過實作子類🄵來擴充*EnvBuilder*，🄵在建立的🄵擬環境中安裝 setuptools 與 pip：

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
```

(繼續下一頁)

(繼續上一頁)

```
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    此建構子會安裝 setuptools 和 pip，使你能🄵使用 pip 或
    easy_install 安裝其他套件到建立的🄵擬環境中。

    :param nodist: 若🄵 true，則不會安裝 setuptools 和 pip 至
                   所建立的🄵擬環境中。
    :param nopip: 若🄵 true，則不會安裝 pip 至所建立的
                  🄵擬環境中。
    :param progress: 若安裝了 setuptools 或 pip，可透過傳入 progress
                     可呼叫物件來監控安裝進度。若有指定，該可呼叫物件會接收兩個引數：
                     一個表示進度的字串，以及一個表示該字串來源的情境。
                     context 引數可能有三種值：
                     'main' 表示該呼叫來自 virtualize() 本身，
                     'stdout' 與 'stderr' 則分🄵表示從子行程輸出資料串流中
                     讀取的輸出🄵容，該子行程用於安裝應用程式。

                     若未指定可呼叫物件，則預設的進度資訊會輸出至 sys.stderr。
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        設定在建立🄵擬環境時需預先安裝的套件。

        :param context: 處理中的🄵擬環境建立請求所需的資訊。
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # 🄵有 setuptools 就無法安裝 pip
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        從子行程的輸出串流中讀取每一列，🄵將其傳遞給進度回呼函式（若有指定），或將進度資訊寫入 sys.
↪stderr。
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
```

(繼續下一頁)

(繼續上一頁)

```
                sys.stderr.write(s.decode('utf-8'))
            sys.stderr.flush()
    stream.close()

def install_script(self, context, name, url):
    _, _, path, _, _, _ = urlparse(url)
    fn = os.path.split(path)[-1]
    binpath = context.bin_path
    distpath = os.path.join(binpath, fn)
    # 將🄵本下載到🄵擬環境的執行檔資料夾中
    urlretrieve(url, distpath)
    progress = self.progress
    if self.verbose:
        term = '\n'
    else:
        term = ''
    if progress is not None:
        progress('Installing %s ...%s' % (name, term), 'main')
    else:
        sys.stderr.write('Installing %s ...%s' % (name, term))
        sys.stderr.flush()
    # 安裝到🄵擬環境中
    args = [context.env_exe, fn]
    p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
    t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
    t1.start()
    t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
    t2.start()
    p.wait()
    t1.join()
    t2.join()
    if progress is not None:
        progress('done.', 'main')
    else:
        sys.stderr.write('done.\n')
    # 清理——不再需要
    os.unlink(distpath)

def install_setuptools(self, context):
    """
    Install setuptools in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = "https://bootstrap.pypa.io/ez_setup.py"
    self.install_script(context, 'setuptools', url)
    # 清除已下載的 setuptools 檔案
    pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:
        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    """
    在🄵擬環境中安裝 pip。

    :param context: 處理中的🄵擬環境建立請求相關資訊。
    """
    url = 'https://bootstrap.pypa.io/get-pip.py'
```

(繼續下一頁)

(繼續上一頁)

```
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates virtual Python '
                                                 'environments in one or '
                                                 'more target '
                                                 'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual "
                             "environment.")
    parser.add_argument('--system-site-packages', default=False,
                        action='store_true', dest='system_site',
                        help='Give the virtual environment access to the '
                             'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than copies, '
                             'when symlinks are not the default for '
                             'the platform.')
    parser.add_argument('--clear', default=False, action='store_true',
                        dest='clear', help='Delete the contents of the '
                                           'virtual environment '
                                           'directory if it already '
                                           'exists, before virtual '
                                           'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                                             'environment directory to '
                                             'use this version of '
                                             'Python, assuming Python '
                                             'has been upgraded '
                                             'in-place.')
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                                             'from the scripts which '
                                             'install setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                 clear=options.clear,
                                 symlinks=options.symlinks,
                                 upgrade=options.upgrade,
                                 nodist=options.nodist,
                                 nopip=options.nopip,
```

(繼續下一頁)

(繼續上一頁)

```
                                    verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

此🄵本也可從🄵上下載。

28.3 zipapp —-- 管理可執行的 Python zip 封存檔案

在 3.5 版被加入.

原始碼： Lib/zipapp.py

本模組提供工具來建立包含 Python 程式碼的 zip 檔案，這些檔案可以 直接由 Python 直譯器執行。此模組同時提供了*zipapp 命令列介面* 和*zipapp Python API*。

28.3.1 基本范例

以下範例展示了如何使用*zipapp 命令列介面*，從一個包含 Python 程式碼的目🄴中建立一個可執行的封存檔案。執行時，該封存檔案將會執行封存檔案中 `myapp` 模組的 `main` 函式。

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.3.2 命令列介面

當從命令列作🄵程式呼叫時，會使用以下格式：

```
$ python -m zipapp source [options]
```

如果 *source* 是一個目🄴，這將會從 *source* 的🄵容建立一個封存檔案。如果 *source* 是一個檔案，它應該是一個封存檔案，🄵且它將被🄵🄵到目標封存檔案（或者如果指定了 --info 選項，將會顯示其 shebang 行的🄵容）。

可使用以下選項：

-o <output>, **--output**=<output>

將輸出寫入名🄵 *output* 的檔案。如果未指定此選項，輸出檔名將與輸入 *source* 相同，🄵加上 `.pyz` 副檔名。如果給定了明確的檔名，則會直接使用它（因此如果需要，應包含 `.pyz` 副檔名）。

如果 *source* 是一個封存檔案，則必須指定輸出檔名（且在這種情🄵下，*output* 不得與 *source* 相同）。

-p <interpreter>, **--python**=<interpreter>

在封存檔案中加入一行 `#!`，指定 *interpreter* 作🄵要執行的命令。此外，在 POSIX 上，使封存檔案成🄵可執行檔。預設是不寫入 `#!` 行，也不讓檔案成🄵可執行檔。

-m <mainfn>, **--main**=<mainfn>

在封存檔案中寫入一個執行 *mainfn* 的 `__main__.py` 檔案。*mainfn* 引數應🄵"pkg.mod:fn" 的形式，其中"pkg.mod" 是封存檔案中的一個套件/模組，而"fn" 是給定模組中的一個可呼叫物件。`__main__.py` 檔案將會執行該可呼叫物件。

🄵🄵封存檔案時，不可指定 *--main*。

-c, --compress

使用 deflate 方法壓縮檔案，以🄵少輸出檔案的大小。預設情🄵下，檔案在封存檔案中是不壓縮儲存的。

🄵🄵封存檔案時，*--compress* 🄵有作用。

在 3.7 版被加入.

--info

顯示嵌入在封存檔案中的直譯器，以供診斷之用。在這種情🄵下，任何其他選項都會被忽略，且 SOURCE 必須是一個封存檔案，而不是一個目🄵。

-h, --help

印出簡短的使用訊息🄵退出。

28.3.3 Python API

此模組定義了兩個便利的函式：

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)

從 *source* 建立一個應用程式封存檔案。source 可以是以下任何一種：

- 一個目🄵的名稱，或是一個指向目🄵的類路徑物件，在這種情🄵下，將會從該目🄵的🄵容建立一個新的應用程式封存檔案。
- 一個現有應用程式封存檔案的名稱，或是一個指向該檔案的類路徑物件，在這種情🄵下，該檔案會被🄵🄵到目標（🄵修改它以反映🄵 *interpreter* 引數所給定的值）。如果需要，檔名應包含 `.pyz` 副檔名。
- 一個以位元組模式開🄵以供讀取的檔案物件。該檔案的🄵容應該是一個應用程式封存檔案，🄵且檔案物件被假定位於封存檔案的開頭。

target 引數🄵定了🄵生的封存檔案將被寫入何處：

- 如果它是一個檔案的名稱，或是一個類路徑物件，封存檔案將被寫入該檔案。
- 如果它是一個開🄵的檔案物件，封存檔案將被寫入該檔案物件，該物件必須以位元組模式開🄵以供寫入。
- 如果省略了 target（或🄵 `None`），則 source 必須是一個目🄵，而 target 將是一個與 source 同名的檔案，🄵加上 `.pyz` 副檔名。

interpreter 引數指定了將用來執行封存檔案的 Python 直譯器名稱。它會作🄵「shebang」行寫入封存檔案的開頭。在 POSIX 上，這將由作業系統解讀，而在 Windows 上，它將由 Python 🄵動器處理。省略 *interpreter* 將導致不寫入 shebang 行。如果指定了直譯器，且 target 是一個檔名，則目標檔案的可執行位元將被設定。

main 引數指定了一個可呼叫物件的名稱，它將被用作封存檔案的主程式。只有當 source 是一個目🄵，且該 source 尚未包含 `__main__.py` 檔案時，才能指定此引數。*main* 引數應🄵用「pkg.module:callable」的形式，封存檔案將透過引入「pkg.module」🄵執行給定的無引數可呼叫物件來執行。如果 source 是一個目🄵且不包含 `__main__.py` 檔案，則省略 *main* 會導致錯誤，因🄵這樣🄵生的封存檔案將無法執行。

可選的 *filter* 引數指定了一個回呼函式，該函式會被傳入一個 Path 物件，代表要加入的檔案路徑（相對於 source 目🄵）。如果要加入該檔案，它應該回傳 `True`。

可選的 *compressed* 引數🄵定檔案是否被壓縮。如果設定🄵 `True`，封存檔案中的檔案將使用 deflate 方法進行壓縮；否則，檔案將不壓縮儲存。🄵🄵現有封存檔案時，此引數無效。

如果🄵 *source* 或 *target* 指定了檔案物件，呼叫者有責任在呼叫 create_archive 後將其關閉。

🄵🄵現有封存檔案時，提供的檔案物件只需要 `read` 和 `readline`，或 `write` 方法。從目🄵建立封存檔案時，如果 target 是一個檔案物件，它將被傳遞給 `zipfile.ZipFile` 類🄵，🄵且必須提供該類🄵所需的方法。

在 3.7 版的變更: 新增 *filter* 與 *compressed* 參數。

`zipapp.`**`get_interpreter`**`(`*archive*`)`

回傳封存檔案開頭 `#!` 行中指定的直譯器。如果🄵有 `#!` 行，則回傳*None*。*archive* 引數可以是一個檔名或一個以位元組模式開🄵以供讀取的類檔案物件。它被假定位於封存檔案的開頭。

28.3.4 范例

將一個目🄵打包成一個封存檔案，🄵執行它。

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

使用*create_archive()* 函式也可以達到同樣的效果：

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

要讓應用程式在 POSIX 上可以直接執行，請指定要使用的直譯器。

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

要替🄵現有封存檔案上的 shebang 行，請使用*create_archive()* 函式建立一個修改過的封存檔案：

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

要原地 (in place) 更新檔案，請使用*BytesIO* 物件在記憶體中進行替🄵，然後再覆寫來源檔案。請注意，原地覆寫檔案存在風險，錯誤可能導致原始檔案遺失。這段程式碼🄵有防範此類錯誤，但生🄵環境的程式碼應作此類防護。此外，此方法只有在封存檔案能容納於記憶體中時才有效：

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.3.5 指定直譯器

請注意，如果你指定了一個直譯器然後分發你的應用程式封存檔案，你需要確保所使用的直譯器是可🄵的。Windows 的 Python 🄵動器支援大多數常見形式的 POSIX `#!` 行，但還有其他問題需要考慮：

- 如果你使用「/usr/bin/env python」（或「python」命令的其他形式，例如「/usr/bin/python」），你需要考慮到你的使用者可能將 Python 2 或 Python 3 作🄵預設版本，🄵編寫你的程式碼使其能在兩個版本下運作。
- 如果你使用一個明確的版本，例如「/usr/bin/env python3」，你的應用程式將無法🄵🄵有該版本的使用者運作。（如果你尚未讓你的程式碼與 Python 2 相容，這可能正是你想要的）。
- 🄵有辦法🄵「python X.Y 或更新版本」，所以要小心使用像「/usr/bin/env python3.4」這樣的確切版本，因🄵舉例來🄵，你將需要🄵 Python 3.5 的使用者更改你的 shebang 行。

通常，你應該使用「/usr/bin/env python2」或「/usr/bin/env python3」，這取🄵於你的程式碼是🄵 Python 2 還是 Python 3 編寫的。

28.3.6 使用 zipapp 建立獨立應用程式

使用*zipapp*模組，可以建立獨立的 Python 程式，這些程式可以分發給終端使用者，他們只需要在他們的系統上安裝合適版本的 Python。實現這一點的關鍵是將應用程式的所有依賴項與應用程式程式碼一起打包到封存檔案中。

建立獨立封存檔案的步驟如下：

1. 像平常一樣在一個目🄵中建立你的應用程式，這樣你就有一個 `myapp` 目🄵，其中包含一個 `__main__.py` 檔案和任何支援的應用程式程式碼。
2. 使用 pip 將你應用程式的所有依賴項安裝到 `myapp` 目🄵中：

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   （這假設你的專案需求在一個 `requirements.txt` 檔案中——如果🄵有，你可以在 pip 命令列上手動列出依賴項）。
3. 使用以下命令打包應用程式：

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

這將🄵生一個獨立的可執行檔，可以在任何有適當直譯器的機器上執行。詳情請參🄵*指定直譯器*。它可以作🄵單一檔案交付給使用者。

在 Unix 上，`myapp.pyz` 檔案本身就是可執行的。如果你偏好一個「普通」的命令名稱，可以重新命名檔案以移除 `.pyz` 副檔名。在 Windows 上，`myapp.pyz[w]` 檔案是可執行的，因🄵 Python 直譯器在安裝時會🄵🄵 `.pyz` 和 `.pyzw` 副檔名。

注意事項

如果你的應用程式依賴於一個包含 C 擴充的套件，該套件無法從 zip 檔案中執行（這是一個作業系統的限制，因🄵可執行程式碼必須存在於檔案系統中，作業系統的載入器才能載入它）。在這種情🄵下，你可以從 zipfile 中排除該依賴項，🄵要求你的使用者安裝它，或者將它與你的 zipfile 一起發布，🄵在你的 `__main__.py` 中加入程式碼，將包含解壓縮後模組的目🄵加入到 `sys.path` 中。在這種情🄵下，你需要確保🄵你的目標架構提供適當的二進位檔案（🄵可能在 runtime 根據使用者的機器選擇正確的版本加入到 `sys.path` 中）。

28.3.7 Python Zip 應用程式封存檔案格式

自 2.6 版以來，Python 就能🄵執行包含 `__main__.py` 檔案的 zip 檔案。🄵了能被 Python 執行，一個應用程式封存檔案只需要是一個標準的 zip 檔案，其中包含一個 `__main__.py` 檔案，該檔案將作🄵應用程式的進入點執行。如同任何 Python 🄵本一樣，🄵本的父目🄵（在此情🄵下是 zip 檔案）將被放置在*sys.path*上，因此可以從 zip 檔案中引入更多的模組。

zip 檔案格式允許在 zip 檔案前附加任意資料。zip 應用程式格式利用此功能在檔案前附加一個標準的 POSIX「shebang」行（`#!/path/to/interpreter`）。

因此，Python zip 應用程式格式的正式定義如下：

1. 一個可選的 shebang 行，包含字元 `b'#!'`，後跟一個直譯器名稱，然後是一個🄵行符（`b'\n'`）。直譯器名稱可以是任何作業系統「shebang」處理或 Windows 上的 Python 🄵動器可接受的🄵容。在 Windows 上，直譯器應以 UTF-8 編碼，在 POSIX 上應以*sys.getfilesystemencoding()* 編碼。
2. 由*zipfile*模組🄵生的標準 zipfile 資料。zipfile 🄵容 * 必須 * 包含一個名🄵 `__main__.py` 的檔案（該檔案必須位於 zipfile 的「根目🄵」中——即，它不能在子目🄵中）。zipfile 資料可以被壓縮或不壓縮。

如果應用程式封存檔案有 shebang 行，它在 POSIX 系統上可能會設定可執行位元，以允許其直接執行。

🄵有要求必須使用此模組中的工具來建立應用程式封存檔案——此模組是一個便利工具，但以任何方式建立的符合上述格式的封存檔案對 Python 都是可接受的。

CHAPTER 29

Python Runtime 服務

本章節所描述的模組提供了與 Python 直譯器及其與環境互動相關的廣泛服務。以下是概觀：

29.1 sys --- 系統特定的參數與函式

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. Unless explicitly noted otherwise, all variables are read-only.

sys.**abiflags**

On POSIX systems where Python was built with the standard `configure` script, this contains the ABI flags as specified by **PEP 3149**.

在 3.2 版被加入.

在 3.8 版的變更: Default flags became an empty string (`m` flag for pymalloc has been removed).

可用性: Unix.

sys.**addaudithook**(*hook*)

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

When an auditing event is raised through the *sys.audit()* function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by `PySys_AddAuditHook()` are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a "sandbox". In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as *ctypes*) should be completely removed or closely monitored.

呼叫*sys.addaudithook()* 本身會引發一個不帶任何引數、名🄵 `sys.addaudithook` 的稽核事件。如果任何現有的 hook 引發從*RuntimeError* 衍生的例外，則不會添加新的 hook 🄵抑制🄴常。因此，除非呼叫者控制所有已存在的 hook，他們不能假設他們的 hook 已被添加。

所有會被 CPython 所引發的事件請參考稽核事件總表、設計相關討論請見 **PEP 578**。

在 3.8 版被加入.

在 3.8.1 版的變更: Exceptions derived from *Exception* but not *RuntimeError* are no longer suppressed.

CPython 實作細節： When tracing is enabled (see *settrace()*), Python hooks are only traced if the callable has a `__cantrace__` member that is set to a true value. Otherwise, trace functions will skip the hook.

`sys.`**`argv`**

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `'-c'`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the *fileinput* module.

另請參🄵*sys.orig_argv*。

> **備🄵**
>
> On Unix, command line arguments are passed by bytes from OS. Python decodes them with filesystem encoding and "surrogateescape" error handler. When you need original bytes, you can get it by `[os.fsencode(arg) for arg in sys.argv]`.

`sys.`**`audit`**`(`*event*, **args*`)`

Raise an auditing event and trigger any active auditing hooks. *event* is a string identifying the event, and *args* may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

舉例來🄵，一個名🄵 `os.chdir` 的稽核事件擁有一個引數 *path*，其🄵容🄵所要求的新工作目🄵。

sys.audit() will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Hooks are added using the *sys.addaudithook()* or `PySys_AddAuditHook()` functions.

The native equivalent of this function is `PySys_Audit()`. Using the native function is preferred when possible.

所有會被 CPython 所引發的事件請參考稽核事件總表。

在 3.8 版被加入.

`sys.`**`base_exec_prefix`**

Equivalent to *exec_prefix*, but referring to the base Python installation.

When running under 🄵擬環境, *exec_prefix* gets overwritten to the virtual environment prefix. *base_exec_prefix*, conversely, does not change, and always points to the base Python installation. Refer to 🄵擬環境 for more information.

在 3.3 版被加入.

`sys.`**`base_prefix`**

Equivalent to *prefix*, but referring to the base Python installation.

When running under *virtual environment*, *prefix* gets overwritten to the virtual environment prefix. *base_prefix*, conversely, does not change, and always points to the base Python installation. Refer to 🄵擬環境 for more information.

在 3.3 版被加入.

`sys.`**`byteorder`**

An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

sys.**builtin_module_names**

A tuple of strings containing the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way --- `modules.keys()` only lists the imported modules.)

另請參🄵*sys.stdlib_module_names* 清單。

sys.**call_tracing**(*func*, *args*)

Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug or profile some other code.

Tracing is suspended while calling a tracing function set by *settrace()* or *setprofile()* to avoid infinite recursion. `call_tracing()` enables explicit recursion of the tracing function.

sys.**copyright**

A string containing the copyright pertaining to the Python interpreter.

sys.**_clear_type_cache**()

Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

在 3.13 版之後被🄵用: Use the more general *_clear_internal_caches()* function instead.

sys.**_clear_internal_caches**()

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

在 3.13 版被加入.

sys.**_current_frames**()

Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the *traceback* module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

引發一個不附帶引數的稽核事件 `sys._current_frames`。

sys.**_current_exceptions**()

Return a dictionary mapping each thread's identifier to the topmost exception currently active in that thread at the time the function is called. If a thread is not currently handling an exception, it is not included in the result dictionary.

This is most useful for statistical profiling.

This function should be used for internal and specialized purposes only.

引發一個不附帶引數的稽核事件 `sys._current_exceptions`。

在 3.12 版的變更: Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

sys.**breakpointhook**()

This hook function is called by built-in *breakpoint()*. By default, it drops you into the *pdb* debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. `pdb.set_trace()`) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in `breakpoint()` function passes its `*args` and `**kws` straight through. Whatever `breakpointhooks()` returns is returned from `breakpoint()`.

The default implementation first consults the environment variable `PYTHONBREAKPOINT`. If that is set to `"0"` then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, `pdb.set_trace()` is called. Otherwise this variable should name a function to run, using Python's dotted-import nomenclature, e.g. `package.subpackage.module.function`. In this case, `package.subpackage.module` would be imported and the resulting module must have a callable named `function()`. This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in *breakpoint()* function.

Note that if anything goes wrong while importing the callable named by `PYTHONBREAKPOINT`, a *RuntimeWarning* is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, `PYTHONBREAKPOINT` is *not* consulted.

在 3.7 版被加入.

`sys._debugmallocstats()`

Print low-level information to stderr about the state of CPython's memory allocator.

If Python is built in debug mode (`configure --with-pydebug option`), it also performs some expensive internal consistency checks.

在 3.3 版被加入.

CPython 實作細節： This function is specific to CPython. The exact output format is not defined here, and may change.

`sys.dllhandle`

Integer specifying the handle of the Python DLL.

可用性: Windows.

`sys.displayhook(value)`

If *value* is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves *value* in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably `'strict'`), encode it to `sys.stdout.encoding` with `'backslashreplace'` error handler.

`sys.displayhook` is called on the result of evaluating an *expression* entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

在 3.2 版的變更: Use `'backslashreplace'` error handler on *UnicodeEncodeError*.

sys.**dont_write_bytecode**

If this is true, Python won't try to write `.pyc` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

sys.**_emscripten_info**

A *named tuple* holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

_emscripten_info.**emscripten_version**

Emscripten version as tuple of ints (major, minor, micro), e.g. `(3, 1, 8)`.

_emscripten_info.**runtime**

運行環境字串，例如🄵覽器使用者代理 (browser user agent) `'Node.js v14.18.2'` 或 `'UNKNOWN'`。

_emscripten_info.**pthreads**

`True` if Python is compiled with Emscripten pthreads support.

_emscripten_info.**shared_memory**

`True` if Python is compiled with shared memory support.

可用性: Emscripten.

在 3.11 版被加入.

sys.**pycache_prefix**

If this is set (not `None`), Python will write bytecode-cache `.pyc` files to (and read them from) a parallel directory tree rooted at this directory, rather than from `__pycache__` directories in the source code tree. Any `__pycache__` directories in the source code tree will be ignored and new `.pyc` files written within the pycache prefix. Thus if you use *compileall* as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

A relative path is interpreted relative to the current working directory.

This value is initially set based on the value of the `-X pycache_prefix=PATH` command-line option or the `PYTHONPYCACHEPREFIX` environment variable (command-line takes precedence). If neither are set, it is `None`.

在 3.8 版被加入.

sys.**excepthook**(*type*, *value*, *traceback*)

This function prints out a given traceback and exception to `sys.stderr`.

When an exception other than *SystemExit* is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

Raise an auditing event `sys.excepthook` with arguments `hook`, `type`, `value`, `traceback` when an uncaught exception occurs. If no hook has been set, `hook` may be `None`. If any hook raises an exception derived from *RuntimeError* the call to the hook will be suppressed. Otherwise, the audit hook exception will be reported as unraisable and `sys.excepthook` will be called.

也參考

The *sys.unraisablehook()* function handles unraisable exceptions and the *threading.excepthook()* function handles exception raised by *threading.Thread.run()*.

sys.**__breakpointhook__**

sys.**__displayhook__**

sys.**__excepthook__**

sys.**__unraisablehook__**

These objects contain the original values of `breakpointhook`, `displayhook`, `excepthook`, and `unraisablehook` at the start of the program. They are saved so that `breakpointhook`, `displayhook` and `excepthook`, `unraisablehook` can be restored in case they happen to get replaced with broken or alternative objects.

在 3.7 版被加入: __breakpointhook__

在 3.8 版被加入: __unraisablehook__

sys.**exception**()

This function, when called while an exception handler is executing (such as an `except` or `except*` clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns `None`.

在 3.11 版被加入.

sys.**exc_info**()

This function returns the old-style representation of the handled exception. If an exception `e` is currently handled (so *`exception()`* would return `e`), *`exc_info()`* returns the tuple `(type(e), e, e.__traceback__)`. That is, a tuple containing the type of the exception (a subclass of *`BaseException`*), the exception itself, and a traceback object which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three `None` values.

在 3.11 版的變更: The `type` and `traceback` fields are now derived from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to *`exc_info()`*.

sys.**exec_prefix**

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also `'/usr/local'`. This can be set at build time with the `--exec-prefix` argument to the **configure** script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory *exec_prefix*`/lib/python`*X.Y*`/config`, and shared library modules are installed in *exec_prefix*`/lib/python`*X.Y*`/lib-dynload`, where *X.Y* is the version number of Python, for example `3.2`.

> **備🄵**
>
> If a *virtual environment* is in effect, this *`exec_prefix`* will point to the virtual environment. The value for the Python installation will still be available, via *`base_exec_prefix`*. Refer to 🄵擬環境 for more information.

在 3.14 版的變更: When running under a *virtual environment*, *`prefix`* and *`exec_prefix`* are now set to the virtual environment prefix by the *path initialization*, instead of *`site`*. This means that *`prefix`* and *`exec_prefix`* always point to the virtual environment, even when *`site`* is disabled (`-S`).

sys.**executable**

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, *`sys.executable`* will be an empty string or `None`.

sys.**exit**([*arg*])

Raise a *`SystemExit`* exception, signaling an intention to exit the interpreter.

The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0--127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, `None` is equivalent to passing zero, and any other object is printed to *stderr* and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since *exit()* ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

在 3.6 版的變更: If an error occurs in the cleanup after the Python interpreter has caught *SystemExit* (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

`sys.`**`flags`**

The *named tuple* *flags* exposes the status of command line flags. Flags should only be accessed only by name and not by index. The attributes are read only.

flags.**debug**	-d
flags.**inspect**	-i
flags.**interactive**	-i
flags.**isolated**	-I
flags.**optimize**	-O 或 -OO
flags.**dont_write_bytecode**	-B
flags.**no_user_site**	-s
flags.**no_site**	-S
flags.**ignore_environment**	-E
flags.**verbose**	-v
flags.**bytes_warning**	-b
flags.**quiet**	-q
flags.**hash_randomization**	-R
flags.**dev_mode**	-X dev （*Python 開發模式*）
flags.**utf8_mode**	-X utf8
flags.**safe_path**	-P
flags.**int_max_str_digits**	-X int_max_str_digits (*integer string conversion length limitation*)
flags.**warn_default_encoding**	-X warn_default_encoding
flags.**gil**	-X gil and PYTHON_GIL
flags.**thread_inherit_context**	-X thread_inherit_context and PYTHON_THREAD_INHERIT_CONTEXT
flags.**context_aware_warnings**	-X context_aware_warnings and PYTHON_CONTEXT_AWARE_WARNINGS

在 3.2 版的變更: 新增 `quiet` 屬性，用於新的 `-q` 旗標。

在 3.2.3 版被加入: `hash_randomization` 屬性。

在 3.3 版的變更: 移除過時的 `division_warning` 屬性。

在 3.4 版的變更: 新增 `isolated` 屬性，用於 `-I isolated` 旗標。

在 3.7 版的變更: Added the `dev_mode` attribute for the new *Python Development Mode* and the `utf8_mode` attribute for the new `-X utf8` flag.

在 3.10 版的變更: 新增 `warn_default_encoding` 屬性，用於 `-X warn_default_encoding` 旗標。

在 3.11 版的變更: 新增 `safe_path` 屬性，用於 `-P` 選項。

在 3.11 版的變更: 新增 `int_max_str_digits` 屬性。

在 3.13 版的變更: 新增 `gil` 屬性。

在 3.14 版的變更: 新增 `thread_inherit_context` 屬性。

在 3.14 版的變更: 新增 `context_aware_warnings` 屬性。

`sys.`**`float_info`**

A *named tuple* holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the 'C' programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], 'Characteristics of floating types', for details.

表格 1: Attributes of the `float_info` named tuple

屬性	float.h macro	解釋
`float_info.`**`epsilon`**	`DBL_EPSILON`	difference between 1.0 and the least value greater than 1.0 that is representable as a float. 另請參🄵*`math.ulp()`*。
`float_info.`**`dig`**	`DBL_DIG`	The maximum number of decimal digits that can be faithfully represented in a float; see below.
`float_info.`**`mant_dig`**	`DBL_MANT_DIG`	Float precision: the number of base-`radix` digits in the significand of a float.
`float_info.`**`max`**	`DBL_MAX`	The maximum representable positive finite float.
`float_info.`**`max_exp`**	`DBL_MAX_EXP`	The maximum integer *e* such that `radix**(e-1)` is a representable finite float.
`float_info.`**`max_10_exp`**	`DBL_MAX_10_EXP`	The maximum integer *e* such that `10**e` is in the range of representable finite floats.
`float_info.`**`min`**	`DBL_MIN`	The minimum representable positive *normalized* float. Use *`math.ulp(0.0)`* to get the smallest positive *denormalized* representable float.
`float_info.`**`min_exp`**	`DBL_MIN_EXP`	The minimum integer *e* such that `radix**(e-1)` is a normalized float.
`float_info.`**`min_10_exp`**	`DBL_MIN_10_EXP`	The minimum integer *e* such that `10**e` is a normalized float.
`float_info.`**`radix`**	`FLT_RADIX`	The radix of exponent representation.
`float_info.`**`rounds`**	`FLT_ROUNDS`	An integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system `FLT_ROUNDS` macro at interpreter startup time: • `-1`: indeterminable • `0`: toward zero • `1`: to nearest • `2`: toward positive infinity • `3`: toward negative infinity All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

The attribute *`sys.float_info.dig`* needs further explanation. If `s` is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
```

(繼續下一頁)

(繼續上一頁)

```
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than *sys.float_info.dig* significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.**float_repr_style**

A string indicating how the *repr()* function behaves for floats. If the string has value `'short'` then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value `'legacy'` and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1.

在 3.1 版被加入.

sys.**getallocatedblocks**()

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call; you may have to call *_clear_internal_caches()* and *gc.collect()* to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, *getallocatedblocks()* is allowed to return 0 instead.

在 3.4 版被加入.

sys.**getunicodeinternedsize**()

Return the number of unicode objects that have been interned.

在 3.12 版被加入.

sys.**getandroidapilevel**()

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see *platform.android_ver()*.

可用性: Android.

在 3.7 版被加入.

sys.**getdefaultencoding**()

Return `'utf-8'`. This is the name of the default string encoding, used in methods like *str.encode()*.

sys.**getdlopenflags**()

Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the *os* module (`RTLD_`*xxx* constants, e.g. *os.RTLD_LAZY*).

可用性: Unix.

sys.**getfilesystemencoding**()

Get the *filesystem encoding*: the encoding used with the *filesystem error handler* to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from *getfilesystemencodeerrors()*.

For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system's preferred representation.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

在 3.2 版的變更: *getfilesystemencoding()* 的結果不再🄵 `None`。

在 3.6 版的變更: Windows is no longer guaranteed to return `'mbcs'`. See **PEP 529** and *_enablelegacywindowsfsencoding()* for more information.

在 3.7 版的變更: Return `'utf-8'` if the *Python UTF-8 Mode* is enabled.

`sys.`**`getfilesystemencodeerrors`**`()`

Get the *filesystem error handler*: the error handler used with the *filesystem encoding* to convert between Unicode filenames and bytes filenames. The filesystem encoding is returned from *getfilesystemencoding()*.

os.fsencode() and *os.fsdecode()* should be used to ensure that the correct encoding and errors mode are used.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

在 3.6 版被加入.

`sys.`**`get_int_max_str_digits`**`()`

Returns the current value for the *integer string conversion length limitation*. See also *set_int_max_str_digits()*.

在 3.11 版被加入.

`sys.`**`getrefcount`**`(`*object*`)`

Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to *getrefcount()*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

CPython 實作細節： *Immortal* objects with a large reference count can be identified via *_is_immortal()*.

在 3.12 版的變更: Immortal objects have very large refcounts that do not match the actual number of references to the object.

`sys.`**`getrecursionlimit`**`()`

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by *setrecursionlimit()*.

`sys.`**`getsizeof`**`(`*object*`[,` *default* `])`

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a *TypeError* will be raised.

getsizeof() calls the object's `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe for an example of using *getsizeof()* recursively to find the size of containers and all their contents.

`sys.`**`getswitchinterval`**`()`

Return the interpreter's "thread switch interval" in seconds; see *setswitchinterval()*.

在 3.2 版被加入.

sys.**_getframe**([*depth*])

Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, *ValueError* is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

引發一個附帶引數 `frame` 的稽核事件 `sys._getframe`。

CPython 實作細節： This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**_getframemodulename**([*depth*])

Return the name of a module from the call stack. If optional integer *depth* is given, return the module that many calls below the top of the stack. If that is deeper than the call stack, or if the module is unidentifiable, `None` is returned. The default for *depth* is zero, returning the module at the top of the call stack.

引發一個附帶引數 `depth` 的稽核事件 `sys._getframemodulename`。

CPython 實作細節： This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

在 3.12 版被加入.

sys.**getobjects**(*limit*[, *type*])

This function only exists if CPython was built using the specialized configure option `--with-trace-refs`. It is intended only for debugging garbage-collection issues.

Return a list of up to *limit* dynamically allocated Python objects. If *type* is given, only objects of that exact type (not subtypes) are included.

Objects from the list are not safe to use. Specifically, the result will include objects from all interpreters that share their object allocator state (that is, ones created with `PyInterpreterConfig.use_main_obmalloc` set to 1 or using `Py_NewInterpreter()`, and the main interpreter). Mixing objects from different interpreters may lead to crashes or other unexpected behavior.

CPython 實作細節： This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

在 3.14 版的變更: The result may include objects from other interpreters.

sys.**getprofile**()

Get the profiler function as set by *setprofile()*.

sys.**gettrace**()

Get the trace function as set by *settrace()*.

CPython 實作細節： The *gettrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.**getwindowsversion**()

Return a named tuple describing the Windows version currently running. The named elements are *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* and *platform_version*. *service_pack* contains a string, *platform_version* a 3-tuple and all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform will be `2` (VER_PLATFORM_WIN32_NT).

product_type may be one of the following values:

Constant	含義
1 (VER_NT_WORKSTATION)	The system is a workstation.
2 (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
3 (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

platform_version returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

備🄵

platform_version derives the version from kernel32.dll which can be of a different version than the OS version. Please use *platform* module for achieving accurate OS version.

可用性: Windows.

在 3.2 版的變更: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

在 3.6 版的變更: 新增 *platform_version*

`sys.`**`get_asyncgen_hooks`**`()`

Returns an *asyncgen_hooks* object, which is similar to a *namedtuple* of the form `(firstiter, finalizer)`, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

在 3.6 版被加入: 更多細節請見 **PEP 525**。

備🄵

This function has been added on a provisional basis (see **PEP 411** for details.)

`sys.`**`get_coroutine_origin_tracking_depth`**`()`

Get the current coroutine origin tracking depth, as set by *set_coroutine_origin_tracking_depth()*.

在 3.7 版被加入.

備🄵

This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

`sys.`**`hash_info`**

A *named tuple* giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see 數值型🄵的雜🄵.

`hash_info.`**`width`**

The width in bits used for hash values

`hash_info.`**`modulus`**

The prime modulus P used for numeric hash scheme

hash_info.**inf**
: The hash value returned for a positive infinity

hash_info.**nan**
: (This attribute is no longer used)

hash_info.**imag**
: The multiplier used for the imaginary part of a complex number

hash_info.**algorithm**
: The name of the algorithm for hashing of str, bytes, and memoryview

hash_info.**hash_bits**
: The internal output size of the hash algorithm

hash_info.**seed_bits**
: The size of the seed key of the hash algorithm

在 3.2 版被加入.

在 3.4 版的變更: 新增 *algorithm*、*hash_bits* 與 *seed_bits*

sys.**hexversion**
: The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in *hex()* function. The *named tuple* *sys.version_info* may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at apiabiversion.

sys.**implementation**
: An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

name is the implementation's identifier, e.g. `'cpython'`. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

version is a named tuple, in the same format as *sys.version_info*. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

hexversion is the implementation version in hexadecimal format, like *sys.hexversion*.

cache_tag is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like `'cpython-33'`. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

supports_isolated_interpreters is a boolean value, whether this implementation supports multiple isolated interpreters. It is `True` for CPython on most platforms. Platforms with this support implement the low-level `_interpreters` module.

> **也參考**
>
> **PEP 684**, **PEP 734**, and *concurrent.interpreters*.

sys.implementation may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, *sys.implementation* will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See **PEP 421** for more information.

在 3.3 版被加入.

在 3.14 版的變更: Added `supports_isolated_interpreters` field.

> **備🄵**
>
> The addition of new required attributes must go through the normal PEP process. See **PEP 421** for more information.

`sys.`**`int_info`**

A *named tuple* that holds information about Python's internal representation of integers. The attributes are read only.

`int_info.`**`bits_per_digit`**

The number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`.

`int_info.`**`sizeof_digit`**

The size in bytes of the C type used to represent a digit.

`int_info.`**`default_max_str_digits`**

The default value for *sys.get_int_max_str_digits()* when it is not otherwise explicitly configured.

`int_info.`**`str_digits_check_threshold`**

The minimum non-zero value for *sys.set_int_max_str_digits()*, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

在 3.1 版被加入.

在 3.11 版的變更: 新增*default_max_str_digits* 和*str_digits_check_threshold*。

`sys.`**`__interactivehook__`**

When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the `PYTHONSTARTUP` file is read, so that you can set this hook there. The *site* module *sets this*.

Raises an *auditing event* `cpython.run_interactivehook` with the hook object as the argument when the hook is called on startup.

在 3.4 版被加入.

`sys.`**`intern`**(*string*)

Enter *string* in the table of "interned" strings and return the interned string -- which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup -- if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not *immortal*; you must keep a reference to the return value of *intern()* around to benefit from it.

sys.**_is_gil_enabled**()

Return *True* if the *GIL* is enabled and *False* if it is disabled.

在 3.13 版被加入.

CPython 實作細節： It is not guaranteed to exist in all implementations of Python.

sys.**is_finalizing**()

Return *True* if the main Python interpreter is *shutting down*. Return *False* otherwise.

另請參🄵*PythonFinalizationError* 例外。

在 3.5 版被加入.

sys.**_jit**

Utilities for observing just-in-time compilation.

CPython 實作細節： JIT compilation is an *experimental implementation detail* of CPython. `sys._jit` is not guaranteed to exist or behave the same way in all Python implementations, versions, or build configurations.

在 3.14 版被加入.

_jit.**is_available**()

Return `True` if the current Python executable supports JIT compilation, and `False` otherwise. This can be controlled by building CPython with the `--experimental-jit` option on Windows, and the `--enable-experimental-jit` option on all other platforms.

_jit.**is_enabled**()

Return `True` if JIT compilation is enabled for the current Python process (implies *sys._jit.is_available()*), and `False` otherwise. If JIT compilation is available, this can be controlled by setting the `PYTHON_JIT` environment variable to `0` (disabled) or `1` (enabled) at interpreter startup.

_jit.**is_active**()

Return `True` if the topmost Python frame is currently executing JIT code (implies *sys._jit.is_enabled()*), and `False` otherwise.

> **備🄵**
>
> This function is intended for testing and debugging the JIT itself. It should be avoided for any other purpose.

> **備🄵**
>
> Due to the nature of tracing JIT compilers, repeated calls to this function may give surprising results. For example, branching on its return value will likely lead to unexpected behavior (if doing so causes JIT code to be entered or exited):

```
>>> for warmup in range(BIG_NUMBER):
...     # This line is "hot", and is eventually JIT-compiled:
...     if sys._jit.is_active():
...         # This line is "cold", and is run in the interpreter:
...         assert sys._jit.is_active()
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    assert sys._jit.is_active()
           ~~~~~~~~~~~~~~~~~~^^
AssertionError
```

sys.**last_exc**

This variable is not always defined; it is set to the exception instance when an exception is not handled and the interpreter prints an error message and a stack traceback. Its intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see *pdb* module for more information.)

在 3.12 版被加入.

sys.**_is_immortal**(*op*)

Return *True* if the given object is *immortal*, *False* otherwise.

> **備🄵**
>
> Objects that are immortal (and thus return `True` upon being passed to this function) are not guaranteed to be immortal in future versions, and vice versa for mortal objects.

在 3.14 版被加入.

CPython 實作細節： This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**_is_interned**(*string*)

Return *True* if the given string is ”interned”, *False* otherwise.

在 3.13 版被加入.

CPython 實作細節： It is not guaranteed to exist in all implementations of Python.

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

These three variables are deprecated; use *sys.last_exc* instead. They hold the legacy representation of `sys.last_exc`, as returned from *exc_info()* above.

sys.**maxsize**

An integer giving the maximum value a variable of type `Py_ssize_t` can take. It's usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform.

sys.**maxunicode**

An integer giving the value of the largest Unicode code point, i.e. `1114111` (`0x10FFFF` in hexadecimal).

在 3.3 版的變更: Before **PEP 393**, `sys.maxunicode` used to be either `0xFFFF` or `0x10FFFF`, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.**meta_path**

A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

> **也參考**
>
> ***importlib.abc.MetaPathFinder***
> The abstract base class defining the interface of finder objects on *meta_path*.
>
> ***importlib.machinery.ModuleSpec***
> The concrete class which *find_spec()* should return instances of.

在 3.4 版的變更: *Module specs* were introduced in Python 3.4, by **PEP 451**.

在 3.12 版的變更: Removed the fallback that looked for a `find_module()` method if a *meta_path* entry didn't have a *find_spec()* method.

`sys.`**`modules`**

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use `sys.modules.copy()` or `tuple(sys.modules)` to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

`sys.`**`orig_argv`**

The list of the original command line arguments passed to the Python executable.

The elements of *sys.orig_argv* are the arguments to the Python interpreter, while the elements of *sys.argv* are the arguments to the user's program. Arguments consumed by the interpreter itself will be present in *sys.orig_argv* and missing from *sys.argv*.

在 3.10 版被加入.

`sys.`**`path`**

A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.

By default, as initialized upon program startup, a potentially unsafe path is prepended to *sys.path* (*before* the entries inserted as a result of `PYTHONPATH`):

- `python -m module` command line: prepend the current working directory.
- `python script.py` command line: prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- `python -c code` and `python` (REPL) command lines: prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the `-P` command line option or the `PYTHONSAFEPATH` environment variable.

A program is free to modify this list for its own purposes. Only strings should be added to *sys.path*; all other data types are ignored during import.

也參考

- Module *site* This describes how to use .pth files to extend *sys.path*.

`sys.`**`path_hooks`**

A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

`sys.`**`path_importer_cache`**

A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no finder is found on *sys.path_hooks* then `None` is stored.

Originally specified in **PEP 302**.

`sys.`**`platform`**

A string containing a platform identifier. Known values are:

系統	`platform` 值
AIX	`'aix'`
Android	`'android'`
Emscripten	`'emscripten'`
FreeBSD	`'freebsd'`
iOS	`'ios'`
Linux	`'linux'`
macOS	`'darwin'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
WASI	`'wasi'`

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('sunos'):
    # 這🄵會是 SunOS 專屬的程式碼...
```

在 3.3 版的變更: On Linux, *sys.platform* doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`.

在 3.8 版的變更: On AIX, *sys.platform* doesn't contain the major version anymore. It is always `'aix'`, instead of `'aix5'` or `'aix7'`.

在 3.13 版的變更: On Android, *sys.platform* now returns `'android'` rather than `'linux'`.

在 3.14 版的變更: On FreeBSD, *sys.platform* doesn't contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.

也參考

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

The *platform* module provides detailed checks for the system's identity.

`sys.`**`platlibdir`**

Name of the platform-specific library directory. It is used to build the path of standard library and the paths of installed extension modules.

It is equal to `"lib"` on most platforms. On Fedora and SuSE, it is equal to `"lib64"` on 64-bit platforms which gives the following `sys.path` paths (where `X.Y` is the Python `major.minor` version):

- `/usr/lib64/pythonX.Y/`: Standard library (like `os.py` of the *os* module)
- `/usr/lib64/pythonX.Y/lib-dynload/`: C extension modules of the standard library (like the *errno* module, the exact filename is platform specific)
- `/usr/lib/pythonX.Y/site-packages/` (always use `lib`, not *sys.platlibdir*): Third-party modules
- `/usr/lib64/pythonX.Y/site-packages/`: C extension modules of third-party packages

在 3.9 版被加入.

`sys.`**`prefix`**

A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `/usr/local`. This can be set at build time with the `--prefix` argument to the **`configure`** script. See *Installation paths* for derived paths.

> **備🄵**
>
> If a *virtual environment* is in effect, this `prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`. Refer to 🄵擬環境 for more information.

在 3.14 版的變更: When running under a *virtual environment*, `prefix` and `exec_prefix` are now set to the virtual environment prefix by the *path initialization*, instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

`sys.`**`ps1`**

`sys.`**`ps2`**

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

`sys.`**`setdlopenflags`**(*n*)

Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

可用性: Unix.

`sys.`**`set_int_max_str_digits`**(*maxdigits*)

Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

在 3.11 版被加入.

`sys.`**`setprofile`**(*profilefunc*)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *Python 的分析器* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

> **備🄵**
>
> The same tracing mechanism is used for `setprofile()` as `settrace()`. To trace calls with `setprofile()` inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The events have the following meaning:

`'call'`
: A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

`'return'`
: A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

'c_call'
A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

'c_return'
A C function has returned. *arg* is the C function object.

'c_exception'
A C function has raised an exception. *arg* is the C function object.

引發一個不附帶引數的稽核事件 `sys.setprofile`。

`sys.`**`setrecursionlimit`**(*limit*)

Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a *RecursionError* exception is raised.

在 3.5.1 版的變更: A *RecursionError* exception is now raised if the new limit is too low at the current recursion depth.

`sys.`**`setswitchinterval`**(*interval*)

Set the interpreter's thread switch interval (in seconds). This floating-point value determines the ideal duration of the "timeslices" allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system's decision. The interpreter doesn't have its own scheduler.

在 3.2 版被加入.

`sys.`**`settrace`**(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using *settrace()* for each thread being debugged or use *threading.settrace()*.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'` or `'opcode'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself, or to another function which would then be used as the local trace function for the scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

> **備🄵**
>
> Tracing is disabled while calling the trace function (e.g. a function set by `settrace()`). For recursive tracing see *call_tracing()*.

The events have the following meaning:

'call'
A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

'line'
The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace

function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to *`False`* on that frame.

`'return'`
A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

`'exception'`
An exception has occurred. The local trace function is called; *arg* is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

`'opcode'`
The interpreter is about to execute a new opcode (see *`dis`* for opcode details). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting `f_trace_opcodes` to *`True`* on the frame.

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning `frame.f_trace = tracefunc` explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which *`settrace()`* doesn't do. Note that in order for this to work, a global tracing function must have been installed with *`settrace()`* in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

For more information on code and frame objects, refer to types.

引發一個不附帶引數的稽核事件 `sys.settrace`。

CPython 實作細節： The *`settrace()`* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

在 3.7 版的變更: `'opcode'` event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

`sys.`**`set_asyncgen_hooks`**(*[firstiter] [, finalizer]*)

Accepts two optional keyword arguments which are callables that accept an *asynchronous generator iterator* as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

引發一個不附帶引數的稽核事件 `sys.set_asyncgen_hooks_firstiter`。

引發一個不附帶引數的稽核事件 `sys.set_asyncgen_hooks_finalizer`。

Two auditing events are raised because the underlying API consists of two calls, each of which must raise its own event.

在 3.6 版被加入: See **PEP 525** for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py

> **備🄵**
>
> This function has been added on a provisional basis (see **PEP 411** for details.)

`sys.`**`set_coroutine_origin_tracking_depth`**(*depth*)

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be `None`.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, set *depth* to zero.

This setting is thread-specific.

在 3.7 版被加入.

備🄵

This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

`sys.`**`activate_stack_trampoline`**(*backend*, */*)

Activate the stack profiler trampoline *backend*. The only supported backend is `"perf"`.

Stack trampolines cannot be activated if the JIT is active.

可用性: Linux.

在 3.12 版被加入.

也參考

- perf_profiling
- https://perf.wiki.kernel.org

`sys.`**`deactivate_stack_trampoline`**()

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

可用性: Linux.

在 3.12 版被加入.

`sys.`**`is_stack_trampoline_active`**()

Return `True` if a stack profiler trampoline is active.

可用性: Linux.

在 3.12 版被加入.

`sys.`**`remote_exec`**(*pid*, *script*)

Executes *script*, a file containing Python code in the remote process with the given *pid*.

This function returns immediately, and the code will be executed by the target process's main thread at the next available opportunity, similarly to how signals are handled. There is no interface to determine when the code has been executed. The caller is responsible for making sure that the file still exists whenever the remote process tries to read it and that it hasn't been overwritten.

The remote process must be running a CPython interpreter of the same major and minor version as the local process. If either the local or remote interpreter is pre-release (alpha, beta, or release candidate) then the local and remote interpreters must be the same exact version.

When the code is executed in the remote process, an *auditing event* `sys.remote_exec` is raised with the *pid* and the path to the script file. This event is raised in the process that called *`sys.remote_exec()`*.

When the script is executed in the remote process, an *auditing event* `cpython.remote_debugger_script` is raised with the path in the remote process. This event is raised in the remote process, not the one that called *sys.remote_exec()*.

可用性: Unix, Windows.

在 3.14 版被加入.

sys.**_enablelegacywindowsfsencoding**()

Changes the *filesystem encoding and error handler* to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the `PYTHONLEGACYWINDOWSFSENCODING` environment variable before launching Python.

See also *sys.getfilesystemencoding()* and *sys.getfilesystemencodeerrors()*.

可用性: Windows.

> **備🄵**
>
> Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

在 3.6 版被加入: 更多細節請見 **PEP 529**。

Deprecated since version 3.13, will be removed in version 3.16: Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

sys.**stdin**

sys.**stdout**

sys.**stderr**

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to *input()*);
- `stdout` is used for the output of *print()* and *expression* statements and for the prompts of *input()*;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular *text files* like those returned by the *open()* function. Their parameters are chosen as follows:

- The encoding and error handling are is initialized from `PyConfig.stdio_encoding` and `PyConfig.stdio_errors`.

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system *locale encoding* if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the `PYTHONIOENCODING` environment variable before starting Python or by using the new `-X utf8` command line option and `PYTHONUTF8` environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTDIO` is also set.

- When interactive, the `stdout` stream is line-buffered. Otherwise, it is block-buffered like regular text files. The `stderr` stream is line-buffered in both cases. You can make both streams unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

在 3.9 版的變更: Non-interactive `stderr` is now line-buffered instead of fully buffered.

> **備🄵**
>
> To write or read binary data from/to the standard streams, use the underlying binary *buffer* object. For example, to write bytes to *stdout*, use `sys.stdout.buffer.write(b'abc')`.
>
> However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like *io.StringIO* which do not support the `buffer` attribute.

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

> **備🄵**
>
> Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with **pythonw**.

`sys.stdlib_module_names`

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed: pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed: sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

另請參🄵*sys.builtin_module_names* 清單。

在 3.10 版被加入.

`sys.thread_info`

A *named tuple* holding information about the thread implementation.

`thread_info.name`

The name of the thread implementation:

- `"nt"`: Windows 執行緒
- `"pthread"`: POSIX 執行緒
- `"pthread-stubs"`: stub POSIX threads (on WebAssembly platforms without threading support)
- `"solaris"`: Solaris threads

`thread_info.lock`

The name of the lock implementation:

- `"semaphore"`: a lock uses a semaphore

- `"mutex+cond"`: a lock uses a mutex and a condition variable
- 🄵 `None` 表示此資訊未知

`thread_info.`**`version`**

The name and version of the thread library. It is a string, or `None` if this information is unknown.

在 3.3 版被加入.

`sys.`**`tracebacklimit`**

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

`sys.`**`unraisablehook`**(*unraisable*, /)

處理一個不可被引發的例外。

Called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (*`gc.collect()`*).

unraisable 引數具有以下屬性:

- `exc_type`: 例外型🄵。
- `exc_value`: 例外值，可以🄵 `None`。
- `exc_traceback`: 例外追🄵，可以🄵 `None`。
- `err_msg`: 錯誤訊息，可以🄵 `None`。
- `object`: 導致例外的物件，可以🄵 `None`。

The default hook formats `err_msg` and `object` as: `f'{err_msg}: {object!r}'`; use "Exception ignored in" error message if `err_msg` is `None`.

`sys.unraisablehook()` can be overridden to control how unraisable exceptions are handled.

也參考

處理未被捕捉到例外的*`excepthook()`*。

警告

Storing `exc_value` using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing `object` using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing `object` after the custom hook completes to avoid resurrecting objects.

Raise an auditing event `sys.unraisablehook` with arguments *hook*, *unraisable* when an exception that cannot be handled occurs. The *unraisable* object is the same as what will be passed to the hook. If no hook has been set, *hook* may be `None`.

在 3.8 版被加入.

`sys.`**`version`**

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use *`version_info`* and the functions provided by the *`platform`* module.

sys.**api_version**

The C API version, equivalent to the C macro `PYTHON_API_VERSION`. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

sys.**version_info**

A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

在 3.1 版的變更: 新增了附名的元件屬性。

sys.**warnoptions**

This is an implementation detail of the warnings framework; do not modify this value. Refer to the *warnings* module for more information on the warnings framework.

sys.**winver**

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python.

可用性: Windows.

sys.**monitoring**

Namespace containing functions and constants for register callbacks and controlling monitoring events. See *sys.monitoring* for details.

sys.**_xoptions**

A dictionary of the various implementation-specific flags passed through the `-X` command-line option. Option names are either mapped to their values, if given explicitly, or to *True*. Example:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython 實作細節： This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

在 3.2 版被加入.

引用

29.2 `sys.monitoring` --- 執行事件監控

在 3.12 版被加入.

備🄵

sys.monitoring 是*sys* 模組🄴的一個命名空間，不是一個獨立的模組，所以不需要 `import sys.monitoring`，只需 `import sys` 然後使用 `sys.monitoring`。

此命名空間提供對🄴動和控制事件監控所需的函式和常數的存取。

當程式執行時，會發生一些能被監控工具關注的事件。*sys.monitoring* 命名空間提供了在發生欲關注事件時接收回呼的方法。

監控 API 由三個元件組成：

- *工具識🄵器*
- *事件*
- *回呼 (callbacks)*

29.2.1 工具識🄵器

工具識🄵器是一個整數和關聯的名稱。工具識🄴器用於防止工具相互干擾🄴允許多個工具同時運作。目前工具是完全獨立的，不能用來互相監控。將來此限制可能會取消。

在🄵🄵或🄵動事件之前，工具應選擇一個識🄴器。識🄵器是 0 到 5（含）範圍🄴的整數。

🄵🄵和使用工具

sys.monitoring.**use_tool_id**(*tool_id:* int, *name:* str, /) → *None*

必須在使用 *tool_id* 之前呼叫。*tool_id* 必須在 0 到 5（含）範圍🄴。如果 *tool_id* 正在使用，則引發*ValueError*。

sys.monitoring.**clear_tool_id**(*tool_id:* int, /) → *None*

Unregister all events and callback functions associated with *tool_id*.

sys.monitoring.**free_tool_id**(*tool_id:* int, /) → *None*

一旦工具不再需要 *tool_id* 就應該被呼叫。在釋放 *tool_id* 之前會呼叫*clear_tool_id()*。

sys.monitoring.**get_tool**(*tool_id:* int, /) → *str* | *None*

如果 *tool_id* 正在使用，則回傳該工具的名稱，否則回傳 None。*tool_id* 必須在 0 到 5（含）範圍🄴。

對於事件，🄵擬機器對所有 ID 的處理都是相同的，但預先定義了以下 ID，以使工具間的協作更容易：

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

29.2.2 事件

支援以下事件：

sys.monitoring.events.**BRANCH_LEFT**

一個條件分支往左。

It is up to the tool to determine how to present ”left” and ”right” branches. There is no guarantee which branch is ”left” and which is ”right”, except that it will be consistent for the duration of the program.

sys.monitoring.events.**BRANCH_RIGHT**

一個條件分支往右。

sys.monitoring.events.**CALL**

Python 程式碼中的呼叫（事件發生在呼叫之前）。

sys.monitoring.events.**C_RAISE**

從任何可呼叫物件引發的例外，除 Python 函式外（事件發生在退出之後）。

sys.monitoring.events.**C_RETURN**

從任何可呼叫函式回傳，除 Python 函式外（事件發生在回傳之後）。

sys.monitoring.events.**EXCEPTION_HANDLED**
一個例外被處理。

sys.monitoring.events.**INSTRUCTION**
🄵擬機器指令即將被執行。

sys.monitoring.events.**JUMP**
在控制流程圖中進行無條件跳轉。

sys.monitoring.events.**LINE**
即將執行的指令與前一條指令的列號不同。

sys.monitoring.events.**PY_RESUME**
Python 函式的繼續執行（對於🄵生器和協程函式），除了 `throw()` 呼叫。

sys.monitoring.events.**PY_RETURN**
從 Python 函式回傳（發生在即將回傳之前，被呼叫者的 frame 將位於堆🄵上）。

sys.monitoring.events.**PY_START**
Python 函式的開始（在呼叫後立即發生，被呼叫者的 frame 將位於堆🄵上）

sys.monitoring.events.**PY_THROW**
Python 函式透過 `throw()` 呼叫來繼續。

sys.monitoring.events.**PY_UNWIND**
Exit from a Python function during exception unwinding. This includes exceptions raised directly within the function and that are allowed to continue to propagate.

sys.monitoring.events.**PY_YIELD**
來自 Python 函式的 yield（在即將 yield 之前發生，被呼叫者的 frame 將位於堆🄵上）。

sys.monitoring.events.**RAISE**
例外被引發，除了那些會導致*STOP_ITERATION* 的事件外。

sys.monitoring.events.**RERAISE**
例外被重新引發 (re-raise)，例如在 `finally` 區塊的最後。

sys.monitoring.events.**STOP_ITERATION**
一個人🄵的*StopIteration* 被引發；請參🄵*STOP_ITERATION* 事件。

將來可能會新增更多事件。

這些事件是 `sys.monitoring.events` 命名空間的屬性。每個事件以 2 次方的整數常數表示。要定義一組事件，只要將個🄵事件以 bitwise OR 相接即可。例如，若要指定*PY_RETURN* 和*PY_START* 兩個事件，請使用運算式 `PY_RETURN | PY_START`。

sys.monitoring.events.**NO_EVENTS**
`0` 的🄵名，以便使用者可以進行明確比較，例如：

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

Setting this event deactivates all events.

區域事件

區域事件與程式的正常執行相關，🄵發生在明確定義的位置。所有區域事件都可以被停用。區域事件有：

- *PY_START*
- *PY_RESUME*
- *PY_RETURN*
- *PY_YIELD*

- *CALL*
- *LINE*
- *INSTRUCTION*
- *JUMP*
- *BRANCH_LEFT*
- *BRANCH_RIGHT*
- *STOP_ITERATION*

Deprecated event

- BRANCH

The BRANCH event is deprecated in 3.14. Using *BRANCH_LEFT* and *BRANCH_RIGHT* events will give much better performance as they can be disabled independently.

附屬事件 (ancillary events)

附屬事件可以像其他事件一樣被監控，但由另一個事件所控制：

- *C_RAISE*
- *C_RETURN*

C_RETURN 和*C_RAISE* 事件由*CALL* 事件控制。只有當對應的*CALL* 事件受到監控時，才會看到*C_RETURN* 和*C_RAISE* 事件。

其他事件

其他事件不一定與程式中的特定位置相關聯，也不能單獨停用。

其他可以監控的事件有：

- *PY_THROW*
- *PY_UNWIND*
- *RAISE*
- *EXCEPTION_HANDLED*

STOP_ITERATION 事件

PEP 380 指出從🄴生器或協程回傳值時要引發*StopIteration* 例外。然而，這是一種非常低效的回傳值方式，因此有一些 Python 實作（特🄴是 CPython 3.12+）不會引發例外，除非它對其他程式碼是可見的。

🄴了允許工具去監控真正的例外而不🄴慢🄴生器和協程的速度，提供了*STOP_ITERATION* 事件。與*RAISE* 不同，*STOP_ITERATION* 可以區域停用。

Note that the *STOP_ITERATION* event and the *RAISE* event for a *StopIteration* exception are equivalent, and are treated as interchangeable when generating events. Implementations will favor *STOP_ITERATION* for performance reasons, but may generate a *RAISE* event with a *StopIteration*.

29.2.3 開🄵和關閉事件

🄵了監控一個事件，必須打開它🄵🄵🄵相應的回呼。可以透過將事件設定🄴全域與/或只🄴特定程式碼物件來開🄴或關閉事件。即使事件在全域和區域都被打開，事件也只會觸發一次。

全域設定事件

可以透過修改正在監控的事件集合來全域地控制事件。

sys.monitoring.**get_events**(*tool_id:* int, /) → *int*

回傳代表所有有效事件的 int。

sys.monitoring.**set_events**(*tool_id:* int, *event_set:* int, /) → *None*

🄵動 *event_set* 中設定的所有事件。如果 *tool_id* 🄵未正在被使用，則引發 *ValueError*。

預設🄵有有效事件。

各🄵程式碼物件事件

事件還可以基於各🄵程式碼物件進行控制。下面定義的、接受 *types.CodeType* 的函式應該准備好接受來自 Python 中未定義函式的類似物件（請參🄵 c-api-monitoring）。

sys.monitoring.**get_local_events**(*tool_id:* int, *code:* CodeType, /) → *int*

回傳 *code* 的所有區域事件

sys.monitoring.**set_local_events**(*tool_id:* int, *code:* CodeType, *event_set:* int, /) → *None*

🄵動 *event_set* 中針對 *code* 設定的所有區域事件。如果 *tool_id* 🄵未正在被使用，則引發 *ValueError*。

停用事件

sys.monitoring.**DISABLE**

可以從回呼函式回傳的特殊值，以停用目前程式碼位置的事件。

可透過回呼函式回傳 *sys.monitoring.DISABLE* 來停用特定程式碼位置的區域事件。這不會改變被設定的事件，或相同事件的任何其他程式碼位置。

停用特定位置的事件對於高效能監控非常重要。舉例來🄵，如果除少數斷點外，偵錯器可以停用所有監控，那🄵在偵錯器下執行程式就不會有任何開銷 (overhead)。

sys.monitoring.**restart_events**() → *None*

🄵所有工具🄵用由 *sys.monitoring.DISABLE* 停用的所有事件。

29.2.4 🄵🄵回呼函式

sys.monitoring.**register_callback**(*tool_id:* int, *event:* int, *func:* Callable | None, /) → *Callable* | *None*

使用給定的 *tool_id* 來🄵🄵對 *event* 的可呼叫物件 *func*

如果給定的 *tool_id* 和 *event* 已經🄵🄵了另一個回呼，則會取消🄵🄵🄵回傳。否則 *register_callback()* 會回傳 None。

以 func 引數引發一個稽核事件 *(auditing event)* sys.monitoring.register_callback。

可以透過呼叫 sys.monitoring.register_callback(tool_id, event, None) 來取消🄵🄵函式。

回呼函式可以隨時被🄵🄵和取消🄵🄵。

Callbacks are called only once regardless if the event is turned on both globally and locally. As such, if an event could be turned on for both global and local events by your code then the callback needs to be written to handle either trigger.

回呼函式引數

sys.monitoring.**MISSING**

傳遞給回呼函式的特殊值，表示該呼叫🄵有引數。

當有效事件發生時，已🄵🄵的回呼函式會被呼叫。回傳非 *DISABLE* 物件的回呼函式將不會有任何效果。不同的事件會🄵回呼函式提供不同的引數，如下所示：

- *PY_START* 和 *PY_RESUME*：

```
func(code: CodeType, instruction_offset: int) -> object
```

- *PY_RETURN* 和*PY_YIELD*：

```
func(code: CodeType, instruction_offset: int, retval: object) -> object
```

- *CALL*、*C_RAISE* 和*C_RETURN*（*arg0* 可以特🄵是*MISSING*）：

```
func(code: CodeType, instruction_offset: int, callable: object, arg0: object) -> object
```

  code represents the code object where the call is being made, while *callable* is the object that is about to be called (and thus triggered the event). If there are no arguments, *arg0* is set to *sys.monitoring.MISSING*.

  For instance methods, *callable* will be the function object as found on the class with *arg0* set to the instance (i.e. the `self` argument to the method).

- *RAISE*、*RERAISE*、*EXCEPTION_HANDLED*、*PY_UNWIND*、*PY_THROW* 和*STOP_ITERATION*：

```
func(code: CodeType, instruction_offset: int, exception: BaseException) -> object
```

- *LINE*：

```
func(code: CodeType, line_number: int) -> object
```

- *BRANCH_LEFT*, *BRANCH_RIGHT* and *JUMP*:

```
func(code: CodeType, instruction_offset: int, destination_offset: int) -> object
```

  請注意，*destination_offset* 是程式碼接下來要執行的地方。

- *INSTRUCTION*：

```
func(code: CodeType, instruction_offset: int) -> object
```

29.3 `sysconfig` --- 提供 Python 設定資訊的存取

在 3.2 版被加入.

原始碼： Lib/sysconfig

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

29.3.1 Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using `setuptools`.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

`sysconfig.`**`get_config_vars`**(**args*)

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

sysconfig.**get_config_var**(*name*)

Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

用法範例：

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.3.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*. The schemes are used by package installers to determine where to copy files to.

Python currently supports nine schemes:

- *posix_prefix*: scheme for POSIX platforms like Linux or macOS. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for POSIX platforms, when the *home* option is used. This scheme defines paths located under a specific home prefix.
- *posix_user*: scheme for POSIX platforms, when the *user* option is used. This scheme defines paths located under the user's home directory (*site.USER_BASE*).
- *posix_venv*: scheme for *Python virtual environments* on POSIX platforms; by default it is the same as *posix_prefix*.
- *nt*: scheme for Windows. This is the default scheme used when Python or a component is installed.
- *nt_user*: scheme for Windows, when the *user* option is used.
- *nt_venv*: scheme for *Python virtual environments* on Windows; by default it is the same as *nt*.
- *venv*: a scheme with values from either *posix_venv* or *nt_venv* depending on the platform Python runs on.
- *osx_framework_user*: scheme for macOS, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files ('pure' Python).
- *include*: directory for non-platform-specific header files for the Python C-API.
- *platinclude*: directory for platform-specific header files for the Python C-API.
- *scripts*: directory for script files.
- *data*: directory for data files.

29.3.3 User scheme

This scheme is designed to be the most convenient solution for users that don't have write permission to the global site-packages directory or don't want to install into it.

Files will be installed into subdirectories of *site.USER_BASE* (written as *userbase* hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as *site.USER_SITE*).

posix_user

Path	Installation directory
stdlib	*userbase*/lib/python*X.Y*
platstdlib	*userbase*/lib/python*X.Y*
platlib	*userbase*/lib/python*X.Y*/site-packages
purelib	*userbase*/lib/python*X.Y*/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

nt_user

Path	Installation directory
stdlib	*userbase*\Python*XY*
platstdlib	*userbase*\Python*XY*
platlib	*userbase*\Python*XY*\site-packages
purelib	*userbase*\Python*XY*\site-packages
include	*userbase*\Python*XY*\Include
scripts	*userbase*\Python*XY*\Scripts
data	*userbase*

osx_framework_user

Path	Installation directory
stdlib	*userbase*/lib/python
platstdlib	*userbase*/lib/python
platlib	*userbase*/lib/python/site-packages
purelib	*userbase*/lib/python/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

29.3.4 Home scheme

The idea behind the "home scheme" is that you build and maintain a personal stash of Python modules. This scheme's name is derived from the idea of a "home" directory on Unix, since it's not unusual for a Unix user to make their home directory have a layout similar to `/usr/` or `/usr/local/`. This scheme can be used by anyone, regardless of the operating system they are installing for.

posix_home

Path	Installation directory
stdlib	*home*/lib/python
platstdlib	*home*/lib/python
platlib	*home*/lib/python
purelib	*home*/lib/python
include	*home*/include/python
platinclude	*home*/include/python
scripts	*home*/bin
data	*home*

29.3.5 Prefix scheme

The "prefix scheme" is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is---that's why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of "the system" rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.`*X* rather than `/usr/lib/python2.`*X*.

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.`*X*, but those modules would have to be installed to, say, `/mnt/`*@server*`/export/lib/python2.`*X*.

posix_prefix

Path	Installation directory
stdlib	*prefix*/lib/python*X.Y*
platstdlib	*prefix*/lib/python*X.Y*
platlib	*prefix*/lib/python*X.Y*/site-packages
purelib	*prefix*/lib/python*X.Y*/site-packages
include	*prefix*/include/python*X.Y*
platinclude	*prefix*/include/python*X.Y*
scripts	*prefix*/bin
data	*prefix*

nt

Path	Installation directory
stdlib	*prefix*\Lib
platstdlib	*prefix*\Lib
platlib	*prefix*\Lib\site-packages
purelib	*prefix*\Lib\site-packages
include	*prefix*\Include
platinclude	*prefix*\Include
scripts	*prefix*\Scripts
data	*prefix*

29.3.6 安裝路徑函式

sysconfig provides some functions to determine these installation paths.

sysconfig.**get_scheme_names**()

Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_default_scheme**()

Return the default scheme name for the current platform.

在 3.10 版被加入: This function was previously named `_get_default_scheme()` and considered an implementation detail.

在 3.11 版的變更: When Python runs from a virtual environment, the *venv* scheme is returned.

sysconfig.**get_preferred_scheme**(*key*)

Return a preferred scheme name for an installation layout specified by *key*.

key must be either `"prefix"`, `"home"`, or `"user"`.

The return value is a scheme name listed in *get_scheme_names()*. It can be passed to *sysconfig* functions that take a *scheme* argument, such as *get_paths()*.

在 3.10 版被加入.

在 3.11 版的變更: When Python runs from a virtual environment and `key="prefix"`, the *venv* scheme is returned.

sysconfig.**_get_preferred_schemes**()

Return a dict containing preferred scheme names on the current platform. Python implementers and redistributors may add their preferred schemes to the `_INSTALL_SCHEMES` module-level global value, and modify this function to return those scheme names, to e.g. provide different schemes for system and language package managers to use, so packages installed by either do not mix with those by the other.

End users should not use this function, but *get_default_scheme()* and *get_preferred_scheme()* instead.

在 3.10 版被加入.

sysconfig.**get_path_names**()

Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])

Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary returned by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

如果找不到 *name*，則引發*KeyError*。

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])

Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, `get_paths()` will raise a `KeyError`.

29.3.7 其他函式

sysconfig.**get_python_version**()

回傳 `MAJOR.MINOR` Python 版本號碼字串。類似於 `'%d.%d' % sys.version_info[:2]`。

sysconfig.**get_platform**()

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by `os.uname()`), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn't particularly important.

Examples of returned values:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows will return one of:

- win-amd64 (64-bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win-arm64 (64-bit Windows on ARM64, aka AArch64)
- win32 (all others - specifically, sys.platform is returned)

macOS can return:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

For other non-POSIX platforms, currently just returns `sys.platform`.

sysconfig.**is_python_build**()

Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

sysconfig.**parse_config_h**(*fp*[, *vars*])

Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.**get_config_h_filename**()

回傳 `pyconfig.h` 的路徑。

sysconfig.**get_makefile_filename**()

回傳 `Makefile` 的路徑。

29.3.8 命令列用法

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

29.4 builtins --- 🄵建物件

該模組提供對 Python 所有'🄵建'識🄵符號的直接存取；例如 builtins.open 是🄵建函式*open()* 的全名。

大多數應用程式通常不會顯式地存取此模組，但在提供與🄵建值同名之物件的模組中可能很有用，不過其中還會需要🄵建該名稱。例如，在一個將🄵建*open()* 包裝起來以實現另一版本*open()* 函式的模組中，這個模組可以直接被使用：

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''將輸出轉🄵成大寫的檔案包裝器'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

有個實作細節是，大多數模組都將名稱 __builtins__ 作🄵其全域性變數的一部分以提使用。__builtins__ 的值通常是這個模組或者這個模組的 __dict__ 屬性值。由於這是一個實作細節，因此 Python 的其他實作可能不會使用它。

也參考

- 🄵建常數
- 🄵建的例外
- 🄵建函式
- 🄵建型🄵

29.5 __main__ --- 頂層程式碼環境

在 Python 中，特殊名稱 __main__ 用於兩個重要的建構：

1. 程式頂層環境的名稱，可以使用 __name__ == '__main__' 運算式進行檢查；和
2. 在 Python 套件中的 __main__.py 檔案。

這兩種機制都與 Python 模組有關；使用者如何與它們互動以及它們如何彼此互動。下面會詳細解釋它們。如果你不熟悉 Python 模組，請參🄵教學章節 tut-modules 的介紹。

29.5.1 __name__ == '__main__'

當引入 Python 模組或套件時，__name__ 設定🄵模組的名稱。通常來🄵，這是 Python 檔案本身的名稱，且不含.py 副檔名：

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

如果檔案是套件的一部分，則 __name__ 也會包含父套件 (parent package) 的路徑：

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

但是，如果模組在頂層程式碼環境中執行，則其 __name__ 將被設定🄵字串 '__main__'。

什🄵是「頂層程式碼環境」?

__main__ 是執行頂層程式碼的環境名稱。「頂層程式碼」是使用者指定且第一個開始運作的 Python 模組。它是「頂層」的原因是因🄵它引入程式所需的所有其他模組。有時「頂層程式碼」被稱🄵應用程式的入口點。

頂層程式碼環境可以是：

- 互動式提示字元的作用域：

  ```
  >>> __name__
  '__main__'
  ```

- 將 Python 模組作🄵檔案引數傳遞給 Python 直譯器：

  ```
  $ python helloworld.py
  Hello, world!
  ```

- 使用 -m 引數傳遞給 Python 直譯器的 Python 模組或套件：

  ```
  $ python -m tarfile
  usage: tarfile.py [-h] [-v] (...)
  ```

- Python 直譯器從標准輸入讀取 Python 程式碼：

```
$ echo "import this" | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

- 使用 `-c` 引數傳遞給 Python 直譯器的 Python 程式碼：

```
$ python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

在這些情🄵下，頂層模組的 `__name__` 都會設定🄵 `'__main__'`。

因此，模組可以透過檢查自己的 `__name__` 來發現它是否在頂層環境中執行，這允許當模組未從 import 陳述式初始化時，使用常見的慣用語法 (idiom) 來有條件地執行程式碼：

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

也參考

若要更詳細地了解如何在所有情🄵下設定 `__name__`，請參🄵教學章節 tut-modules。

慣用 (Idiomatic) 用法

某些模組包含僅供🄵本使用的程式碼，例如剖析命令列引數或從標准輸入取得資料。如果從不同的模組匯入這樣的模組（例如對其進行單元測試 (unit test)），則🄵本程式碼也會無意間執行。

這就是使用 `if __name__ == '__main__'` 程式碼區塊派上用場的地方。除非該模組在頂層環境中執行，否則此區塊中的程式碼不會執行。

在 `if __name__ == '__main__'` 下面的區塊中放置盡可能少的陳述式可以提高程式碼的清晰度和正確性。大多數情🄵下，名🄵 `main` 的函式封裝 (encapsulate) 了程式的主要行🄵：

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
    """A dummy wrapper around print."""
    # for demonstration purposes, you can imagine that there is some
    # valuable and reusable logic inside this function
    print(phrase)

def main() -> int:
    """Echo the input arguments to standard output"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # next section explains the use of sys.exit
```

請注意，如果模組🄵有將程式碼封裝在 `main` 函式中，而是直接將其放在 `if __name__ == '__main__'` 區塊中，則 `phrase` 變數對於整個模組來🄵將是全域的。這很容易出錯，因🄵模組中的其他函式可能會無意中使用此全域變數而不是區域變數。`main` 函式解🄵了這個問題。

使用 `main` 函式還有一個額外的好處，`echo` 函式本身是隔離的 (isolated) 🄵且可以在其他地方引入。當引入 `echo.py` 時，`echo` 和 `main` 函式將被定義，但它們都不會被呼叫，因🄵 `__name__ != '__main__'`。

打包時須考慮的事情

`main` 函式通常用於透過將它們指定🄵控制台🄵本的入口點來建立命令列工具。完成後，pip 將函式呼叫插入到模板🄵本中，其中 `main` 的回傳值被傳遞到*`sys.exit()`* 中。例如：

```
sys.exit(main())
```

由於對 `main` 的呼叫包含在*`sys.exit()`* 中，因此期望你的函式將傳回一些可接受作🄵*`sys.exit()`* 輸入的值；通常來🄵，會是一個整數或 `None`（如果你的函式🄵有 return 陳述式，則相當於回傳此值）。

透過我們自己主動遵循這個慣例，我們的模組在直接執行時（即 `python echo.py`）的行🄵，將和我們稍後將其打包🄵 pip 可安裝套件中的控制台🄵本入口點相同。

特🄵是，要謹慎處理從 `main` 函式回傳字串。*`sys.exit()`* 會將字串引數直譯🄵失敗訊息，因此你的程式將有一個表示失敗的結束代碼 `1`，🄵且該字串將被寫入*`sys.stderr`*。前面的 `echo.py` 範例使用慣例的 `sys.exit(main())` 進行示範。

也參考

Python 打包使用者指南包含一系列如何使用現代工具發行和安裝 Python 套件的教學和參考資料。

29.5.2 Python 套件中的 `__main__.py`

如果你不熟悉 Python 套件，請參🄵 tut-packages 的教學章節。最常見的是，`__main__.py` 檔案用於🄵套件提供命令列介面。假設下面有🄵構的套件"bandclass"：

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

當使用 `-m` 旗標 (flag) 直接從命令列呼叫套件本身時，將執行 `__main__.py`。例如：

```
$ python -m bandclass
```

該命令將導致 `__main__.py` 執行。如何利用此機制將取🄵於你正在編寫的套件的性質，但在這種🄵構的情🄵下，允許教師搜尋學生可能是有意義的：

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

請注意，`from .student import search_students` 是相對引入的範例。在引用套件🄵的模組時，可以使用此引入樣式。有關更多詳細資訊，請參🄵 tut-modules 教學章節中的 intra-package-references。

慣用 (Idiomatic) 用法

`__main__.py` 的🄵容通常不會被 `if __name__ == '__main__'` 區塊包圍。相反的，這些檔案保持簡短🄵引入其他模組的函式來執行。那些其他模組就可以輕松地進行單元測試🄵且可以正確地重🄵使用。

如果在套件🄵面的 `__main__.py` 檔案使用 `if __name__ == '__main__'` 區塊，它依然會如預期般地運作。因🄵當引入套件，其 `__name__` 屬性將會包含套件的路徑：

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

但這對於 `.zip` 檔案根目🄵中的 `__main__.py` 檔案不起作用。因此，🄵了保持一致性，最小的、🄵有 `__name__` 檢查的 `__main__.py` 會是首選。

也參考

請參🄵*venv* 作🄵標准函式庫中具有最小 `__main__.py` 的套件🄵範例。它不包含 `if __name__ == '__main__'` 區塊。你可以使用 `python -m venv [directory]` 來呼叫它。

請參🄵*runpy* 取得有關直譯器可執行檔的 `-m` 旗標的更多詳細資訊。

請參🄵*zipapp* 了解如何執行打包成 *.zip* 檔案的應用程式。在這種情🄵下，Python 會在封存檔案的根目🄵中尋找 `__main__.py` 檔案。

29.5.3 `import __main__`

無論 Python 程式是從哪個模組🄵動的，在同一程式中執行的其他模組都可以透過匯入 `__main__` 模組來引入頂層環境的作用域 (*namespace*)。這不會引入 `__main__.py` 檔案，而是引入接收特殊名稱 `'__main__'` 的模組。

這是一個使用 `__main__` 命名空間的範例模組：

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    print(__main__.my_name)
```

該模組的範例用法如下：

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"

def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == "__main__":
    sys.exit(main())
```

現在，如果我們🄴動程式，結果將如下所示：

```
$ python start.py
Define the variable `my_name`!
```

程式的結束代碼將🄵 1，表示出現錯誤。取消🄵解 `my_name = "Dinsdale"` 而修復程式後，現在它以狀態碼 0 結束，表示成功：

```
$ python start.py
Dinsdale
```

請注意，引入 `__main__` 🄵不會因🄵不經意地執行放在 `start` 模組中 `if __name__ == "__main__"` 區塊的頂層程式碼（本來是給🄴本使用的）而造成任何問題。🄵什🄵這樣做會如預期運作?

當 Python 直譯器🄴動時，會在*sys.modules* 中插入一個空的 `__main__` 模組，🄴透過執行頂層程式碼來填充它。在我們的範例中，這是 `start` 模組，它將逐行執行🄴引入 `namely`。接著，`namely` 引入 `__main__`（其實是 `start`）。這就是一個引入循環！幸運的是，由於部分填充的 `__main__` 模組存在於*sys.modules* 中，Python 將其傳遞給 `namely`。請參🄴引入系統參考文件中的關於 __main__ 的特🄴考量了解其工作原理的詳細資訊。

Python REPL 是「頂層環境」的另一個範例，因此 REPL 中定義的任何🄴容都成🄴作域的一部分：

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

`__main__` 作用域用於*pdb* 和*rlcompleter* 的實作。

29.6 `warnings` --- 警告控制

原始碼： Lib/warnings.py

警告訊息通常會在需要提醒使用者程式中的某些狀🄴時發出，而該狀🄴（通常）不至於需要引發例外🄴終止程式。舉例來🄴，當程式使用一個過時的模組時，就可能會需要發出警告。

Python 程式設計師可以透過呼叫此模組中定義的*warn()* 函式來發出警告。（C 語言的程式設計師則使用 `PyErr_WarnEx()`；詳情請參🄵 exceptionhandling）。

警告訊息通常會被寫入*sys.stderr*，但它們的處理方式可以被彈性地更改，從忽略所有警告到將它們轉🄵例外都可以。警告的處理方式會根據警告類🄵、警告訊息的文本，以及發出警告的原始碼位置而有所不同。在同一個原始碼位置重🄵出現的特定警告通常會被抑制。

警告控制有兩個階段：首先，每當發出一個警告時，會先🄵定是否該發出訊息；接著，如果要發出訊息，它會使用一個可由使用者設定的🄵🄵 (hook) 來格式化🄵印出。

是否要發出警告訊息是由警告過濾器 所控制，它是一連串的匹配規則和動作。可以透過呼叫*filterwarnings()* 來新增規則到過濾器中，🄵透過呼叫*resetwarnings()* 將其重設🄵預設狀態。

警告訊息的印出是透過呼叫*showwarning()* 來完成，而它可以被覆寫；此函式的預設實作會透過呼叫*formatwarning()* 來格式化訊息，它也可以被自訂的實作使用。

也參考

logging.captureWarnings() 允許你使用標准的 logging 基礎設施來處理所有警告。

29.6.1 警告類🄵

有許多🄵建的例外代表警告類🄵。這種分類方式對於能🄵過濾掉特定群組的警告很有用。

雖然這些在技術上是🄵*建例外*，但它們被記🄵在這🄵，因🄵從概念上來🄵，它們屬於警告機制的一部分。

使用者程式碼可以透過繼承其中一個標准警告類🄵來定義額外的警告類🄵。一個警告類🄵必須永遠是*Warning* 類🄵的子類🄵。

目前定義了以下警告類🄵：

類🄵	描述
Warning	這是所有警告類🄵的基底類🄵。它是*Exception* 的子類🄵。
UserWarning	*warn()* 的預設類🄵。
DeprecationWarning	關於已🄵用功能的警告的基底類🄵，當這些警告是針對其他 Python 開發者時（預設🄵忽略，除非由 __main__ 中的程式碼觸發）。
SyntaxWarning	Base category for warnings about dubious syntactic features (typically emitted when compiling Python source code, and hence may not be suppressed by runtime filters)
RuntimeWarning	關於可疑 runtime 功能的警告的基底類🄵。
FutureWarning	關於已🄵用功能的警告的基底類🄵，當這些警告是針對以 Python 編寫的應用程式的終端使用者時。
PendingDeprecationWarning	關於未來將被🄵用的功能的警告的基底類🄵（預設🄵忽略）。
ImportWarning	在引入模組過程中觸發的警告的基底類🄵（預設🄵忽略）。
UnicodeWarning	與 Unicode 相關的警告的基底類🄵。
BytesWarning	與*bytes* 和*bytearray* 相關的警告的基底類🄵。
ResourceWarning	與資源使用相關的警告的基底類🄵（預設🄵忽略）。

在 3.7 版的變更: 在過去，*DeprecationWarning* 和*FutureWarning* 是根據一個功能是被完全移除還是改變其行🄵來區分的。它們現在是根據其目標受🄵以及預設警告過濾器處理它們的方式來區分。

29.6.2 警告過濾器

警告過濾器控制警告是被忽略、顯示，還是轉🄵錯誤（引發一個例外）。

從概念上講，警告過濾器維護一個有序的過濾器規格串列；任何特定的警告都會依次與串列中的每個過濾器規格進行比對，直到找到匹配項；過濾器🄵定了匹配項的處理方式。每個條目都是一個 (*action*, *message*, *category*, *module*, *lineno*) 形式的元組，其中：

- *action* 是以下字串之一：

值	處理方式
"default"	🄵發出警告的每個位置（模組 + 行號）印出第一次出現的匹配警告
"error"	將匹配的警告轉🄵例外
"ignore"	永不印出匹配的警告
"always"	總是印出匹配的警告
"all"	"always" 的🄵名
"module"	🄵發出警告的每個模組印出第一次出現的匹配警告（不論行號）
"once"	只印出第一次出現的匹配警告，不論位置

- *message* 是一個包含正規表示式的字串，警告訊息的開頭必須與其匹配（不區分大小寫）。在 `-W` 和 `PYTHONWARNINGS` 中，*message* 是一個字面字串，警告訊息的開頭必須包含該字串（不區分大小寫），🄵忽略 *message* 開頭或結尾的任何空白字元。
- *category* 是一個類🄵（*Warning* 的子類🄵），警告類🄵必須是它的子類🄵才能匹配。
- *module* 是一個包含正規表示式的字串，完整限定模組名稱的開頭必須與其匹配（區分大小寫）。在 `-W` 和 `PYTHONWARNINGS` 中，*module* 是一個字面字串，完整限定模組名稱必須與其相等（區分大小寫），🄵忽略 *module* 開頭或結尾的任何空白字元。
- *lineno* 是一個整數，發出警告的行號必須與其匹配，或者🄵 `0` 以匹配所有行號。

由於*Warning* 類🄵衍生自🄵建的*Exception* 類🄵，要將警告轉🄵錯誤，我們只需引發 `category(message)`。

如果一個警告被回報且不匹配任何已🄵🄵的過濾器，則會套用"default" 動作（因此得名）。

重🄵警告的抑制標準

抑制重🄵警告的過濾器會套用以下標准來判斷一個警告是否被視🄵重🄵：

- `"default"`：只有當 (*message*, *category*, *module*, *lineno*) 都相同時，警告才被視🄵重🄵。
- `"module"`：如果 (*message*, *category*, *module*) 相同，則警告被視🄵重🄵，忽略行號。
- `"once"`：如果 (*message*, *category*) 相同，則警告被視🄵重🄵，忽略模組和行號。

描述警告過濾器

警告過濾器由傳遞給 Python 直譯器命令列的 `-W` 選項和 `PYTHONWARNINGS` 環境變數初始化。直譯器會將所有提供條目的引數未經直譯地儲存在*sys.warnoptions* 中；*warnings* 模組在首次引入時會剖析這些引數（無效選項會被忽略，🄵在向*sys.stderr* 印出一條訊息後）。

個🄵的警告過濾器被指定🄵一系列由冒號分隔的欄位：

```
action:message:category:module:line
```

這些欄位中每一個的含義都如警告過濾器 中所述。當在單一行中列出多個過濾器時（例如 `PYTHONWARNINGS`），個🄵的過濾器會以逗號分隔，且後面列出的過濾器優先於前面列出的過濾器（因🄵它們是從左到右應用的，而最近應用的過濾器優先於較早的過濾器）。

常用的警告過濾器適用於所有警告、特定類🄵的警告，或由特定模組或套件引發的警告。一些範例如下：

```
default                       # 顯示所有警告（即使是預設忽略的警告）
ignore                        # 忽略所有警告
error                         # 將所有警告轉🄵🄵錯誤
error::ResourceWarning        # 將 ResourceWarning 訊息視🄵錯誤
default::DeprecationWarning   # 顯示 DeprecationWarning 訊息
ignore,default:::mymodule     # 只回報由 "mymodule" 觸發的警告
error:::mymodule              # 將 "mymodule" 中的警告轉🄵🄵錯誤
```

預設警告過濾器

預設情🄵下，Python 會安裝數個警告過濾器，這些過濾器可以被 `-W` 命令列選項、`PYTHONWARNINGS` 環境變數以及對*filterwarnings()* 的呼叫所覆寫。

在常規的發行建置中，預設的警告過濾器有以下條目（按優先順序排列）：

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

在 偵錯建置中，預設警告過濾器串列是空的。

在 3.2 版的變更: 除了*PendingDeprecationWarning* 之外，*DeprecationWarning* 現在也預設被忽略。

在 3.7 版的變更: 當直接由 __main__ 中的程式碼觸發時，*DeprecationWarning* 會再次預設顯示。

在 3.7 版的變更: *BytesWarning* 不再出現在預設過濾器串列中，而是在指定 -b 兩次時，透過*sys.warnoptions* 進行設定。

覆寫預設過濾器

以 Python 編寫的應用程式的開發者可能希望預設對其使用者隱藏 所有 Python 層級的警告，🄵且只在執行測試或以其他方式處理應用程式時才顯示它們。用於將過濾器設定傳遞給直譯器的*sys.warnoptions* 屬性可以用作一個標記，以指示是否應停用警告:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

建議 Python 程式碼的測試執行器開發者，應確保在預設情🄵下，🄵受測程式碼顯示 所有警告，可使用如下程式碼:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # 在此行程中更改過濾器
    os.environ["PYTHONWARNINGS"] = "default" # 也會影響子行程
```

最後，建議在 __main__ 以外的命名空間中執行使用者程式碼的互動式 shell 開發者，應確保*DeprecationWarning* 訊息預設🄵可見，可使用如下程式碼（其中 user_ns 是用於執行互動式輸入程式碼的模組）:

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

29.6.3 暫時抑制警告

如果你正在使用的程式碼，你知道它會引發一個警告（例如一個已🄵用的函式），但你不想看到這個警告（即使警告已透過命令列明確設定），那🄵可以使用*catch_warnings* 情境管理器來抑制該警告:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

在情境管理器中，所有警告都將被直接忽略。這允許你在使用已知的已🄵用程式碼時不必看到警告，同時又不會抑制其他可能不知道自己正在使用已🄵用程式碼的程式碼所發出的警告。

> **備🄵**
>
> See 情境管理器的🄵行安全性 for details on the concurrency-safety of the *catch_warnings* context manager when used in programs using multiple threads or async functions.

29.6.4 測試警告

要測試程式碼引發的警告，請使用*catch_warnings* 情境管理器。透過它，你可以暫時改變警告過濾器以方便你的測試。例如，執行以下操作來捕獲所有引發的警告以進行檢查：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # 讓所有警告總是會被觸發。
    warnings.simplefilter("always")
    # 觸發一個警告。
    fxn()
    # 驗證一些事情
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

也可以使用 `error` 而非 `always` 來讓所有警告都變成例外。需要注意的一點是，如果一個警告因🄵 `once/default` 規則已經被引發過，那🄵無論設定什🄵過濾器，這個警告都不會再出現，除非與該警告相關的警告🄵🄵表已被清除。

一旦情境管理器退出，警告過濾器就會恢復到進入情境時的狀態。這可以防止測試在不同測試之間以意想不到的方式更改警告過濾器，從而導致不確定的測試結果。

> **備🄵**
>
> See 情境管理器的🄵行安全性 for details on the concurrency-safety of the *catch_warnings* context manager when used in programs using multiple threads or async functions.

在測試多個會引發同類警告的操作時，重要的是要以一種能確認每個操作都在引發新警告的方式來進行測試（例如，將警告設定🄵引發例外，🄵檢查操作是否引發例外；檢查警告串列的長度在每次操作後是否持續增加；或者在每次新操作前從警告串列中🄵除先前的條目）。

29.6.5 🄵新版相依套件更新程式碼

主要針對 Python 開發者（而非以 Python 編寫的應用程式的終端使用者）的警告類🄵預設會被忽略。

值得注意的是，這個「預設忽略」串列包括*DeprecationWarning*（除了 `__main__` 以外的所有模組），這意味著開發者應該確保在測試他們的程式碼時，讓通常被忽略的警告可見，以便及時收到未來 API 破壞性變更的通知（無論是在標準函式庫還是第三方套件中）。

在理想情🄵下，程式碼會有一個合適的測試套件，而測試執行器會在執行測試時負責隱式地🄵用所有警告（*unittest* 模組提供的測試執行器就是這樣做的）。

在較不理想的情🄵下，可以透過將 `-Wd` 傳遞給 Python 直譯器（這是 `-W default` 的簡寫）或在環境中設定 `PYTHONWARNINGS=default` 來檢查應用程式是否使用了已🄵用的介面。這會🄵所有警告🄵用預設處理，包括那些預設被忽略的警告。要更改對遇到的警告所🄵取的動作，你可以更改傳遞給 `-W` 的引數（例如 `-W error`）。有關更多可能性的詳細資訊，請參🄵 `-W` 旗標。

29.6.6 可用的函式

`warnings.`**`warn`**(*message*, *category=None*, *stacklevel=1*, *source=None*, ***, *skip_file_prefixes=()*)

發出一個警告，或者可能忽略它或引發一個例外。*category* 引數（如果提供）必須是一個警告類🄵；它預設🄵*UserWarning*。或者，*message* 可以是一個*Warning* 實例，在這種情🄵下 *category* 將被忽略，🄵使用 `message.__class__`。在這種情🄵下，訊息文本將是 `str(message)`。如果發出的特定警告被警告過濾器 轉🄵錯誤，此函式會引發一個例外。*stacklevel* 引數可以被以 Python 編寫的包裝函式使用，如下所示：

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

這會讓警告指向 deprecated_api 的呼叫者，而不是 deprecated_api 本身的原始碼（因🄵後者會違背警告訊息的目的）。

skip_file_prefixes 關鍵字引數可用於指示在計算堆🄵層級時應忽略哪些堆🄵框。當你希望警告總是在套件外部的呼叫點出現，而固定的 *stacklevel* 不適用於所有呼叫路徑或難以維護時，這會很有用。如果提供，它必須是一個字串的元組。當提供前綴時，stacklevel 會被隱式地覆寫🄵 max(2, stacklevel)。要讓警告歸因於目前套件外部的呼叫者，你可以這樣寫：

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

這使得警告只會從存在於 example 套件之外的呼叫程式碼中，指向 example.lower.one_way() 和 example.higher.another_way() 的呼叫點。

source（如果提供）是發出*ResourceWarning* 的已銷🄵物件。

在 3.6 版的變更: 新增 *source* 參數。

在 3.12 版的變更: 新增 *skip_file_prefixes*。

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

這是一個*warn()* 功能的低階介面，明確傳入訊息、類🄵、檔名和行號，以及可選的模組名稱和🄵🄵表（應🄵模組的 __warningregistry__ 字典）。模組名稱預設🄵去掉 .py 的檔名；如果🄵有傳入🄵🄵表，警告將永遠不會被抑制。*message* 必須是字串，*category* 必須是*Warning* 的子類🄵，或者 *message* 可以是*Warning* 的實例，在這種情🄵下 *category* 將被忽略。

module_globals（如果提供）應該是發出警告的程式碼所使用的全域命名空間。（此引數用於支援顯示在 zip 檔案或其他非檔案系統引入來源中找到的模組的原始碼）。

source（如果提供）是發出*ResourceWarning* 的已銷🄵物件。

在 3.6 版的變更: 新增 *source* 參數。

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

將警告寫入一個檔案。預設實作會呼叫 formatwarning(message, category, filename, lineno, line) 🄵將結果字串寫入 *file*，預設🄵*sys.stderr*。你可以透過賦值給 warnings.showwarning 來用任何可呼叫物件取代此函式。*line* 是要包含在警告訊息中的一行原始碼；如果未提供 *line*，*showwarning()* 將嘗試讀取由 *filename* 和 *lineno* 指定的行。

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

以標準方式格式化警告。這會回傳一個可能包含嵌入🄵行符🄵以🄵行符結尾的字串。*line* 是要包含在警告訊息中的一行原始碼；如果未提供 *line*，*formatwarning()* 將嘗試讀取由 *filename* 和 *lineno* 指定的行。

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

在警告過濾器規格 串列中插入一個條目。預設情🄵下，條目會插入到最前面；如果 *append* 🄵 true，則會插入到最後面。這會檢查引數的型🄵，編譯 *message* 和 *module* 的正規表示式，🄵將它們作🄵一個元組插入到警告過濾器串列中。如果兩個條目都匹配一個特定的警告，那🄵串列中較靠前的條目會覆寫較後面的條目。省略的引數預設🄵一個能匹配所有🄵容的值。

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

在警告過濾器規格 串列中插入一個簡單的條目。函式參數的含義與*filterwarnings()* 相同，但不需要正規表示式，因🄵只要類🄵和行號匹配，插入的過濾器總是會匹配任何模組中的任何訊息。

warnings.**resetwarnings**()

重設警告過濾器。這會舍🄵所有先前對*filterwarnings()* 的呼叫效果，包括 `-W` 命令列選項和對*simplefilter()* 的呼叫。

@warnings.**deprecated**(*msg*, ***, *category=DeprecationWarning*, *stacklevel=1*)

用來指示一個類🄵、函式或重載 (overload) 已被🄵用的裝飾器。

當此裝飾器應用於一個物件時，在使用該物件時，可能會在 runtime 發出🄵用警告。🄵態型🄵檢查器 也會在使用已🄵用物件時🄵生診斷訊息。

用法：

```
from warnings import deprecated
from typing import overload

@deprecated("改用 B")
class A:
    pass

@deprecated("改用 g")
def f():
    pass

@overload
@deprecated("對 int 的支援已🄵用")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

由 *category* 指定的警告將在使用已🄵用物件時於 runtime 發出。對於函式，這發生在呼叫時；對於類🄵，發生在實例化和建立子類🄵時。如果 *category* 是 `None`，則在 runtime 不會發出警告。*stacklevel* 🄵定了警告發出的位置。如果它是 `1`（預設值），警告會在已🄵用物件的直接呼叫者處發出；如果更高，它會在堆🄵的更上層發出。🄵態型🄵檢查器的行🄵不受 *category* 和 *stacklevel* 引數的影響。

傳遞給裝飾器的🄵用訊息會儲存在被裝飾物件的 `__deprecated__` 屬性中。如果應用於重載，裝飾器必須在*@overload* 裝飾器之後，這樣屬性才會存在於*typing.get_overloads()* 回傳的重載上。

在 3.13 版被加入: 參🄵 **PEP 702**。

29.6.7 可用的情境管理器

class warnings.**catch_warnings**(***, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

一個情境管理器，它會🄵🄵🄵在退出時恢復警告過濾器和*showwarning()* 函式。如果 *record* 引數🄵*False*（預設值），情境管理器在進入時回傳*None*。如果 *record* 🄵*True*，則會回傳一個串列，該串列會由自訂的*showwarning()* 函式（該函式也會抑制對 `sys.stdout` 的輸出）所看到的物件逐步填充。串列中的每個物件都具有與*showwarning()* 的引數同名的屬性。

module 引數接受一個模組，用以取代預設的*warnings* 模組，而該（被傳入）模組的過濾器將會受到保護。此引數主要用於測試*warnings* 模組本身。

如果 *action* 引數不是 `None`，則其余引數會傳遞給*simplefilter()*，就像在進入情境時立即呼叫它一樣。

有關 *category* 和 *lineno* 參數的含義，請參🄵警告過濾器。

> **備🄵**
>
> See 情境管理器的🄵行安全性 for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

在 3.11 版的變更: 新增 *action*、*category*、*lineno* 和 *append* 參數。

29.6.8 情境管理器的🄵行安全性

The behavior of `catch_warnings` context manager depends on the `sys.flags.context_aware_warnings` flag. If the flag is true, the context manager behaves in a concurrent-safe fashion and otherwise not. Concurrent-safe means that it is both thread-safe and safe to use within *asyncio coroutines* and tasks. Being thread-safe means that behavior is predictable in a multi-threaded program. The flag defaults to true for free-threaded builds and false otherwise.

If the `context_aware_warnings` flag is false, then `catch_warnings` will modify the global attributes of the `warnings` module. This is not safe if used within a concurrent program (using multiple threads or using asyncio coroutines). For example, if two or more threads use the `catch_warnings` class at the same time, the behavior is undefined.

If the flag is true, `catch_warnings` will not modify global attributes and will instead use a `ContextVar` to store the newly established warning filtering state. A context variable provides thread-local storage and it makes the use of `catch_warnings` thread-safe.

The *record* parameter of the context handler also behaves differently depending on the value of the flag. When *record* is true and the flag is false, the context manager works by replacing and then later restoring the module's `showwarning()` function. That is not concurrent-safe.

When *record* is true and the flag is true, the `showwarning()` function is not replaced. Instead, the recording status is indicated by an internal property in the context variable. In this case, the `showwarning()` function will not be restored when exiting the context handler.

The `context_aware_warnings` flag can be set the `-X context_aware_warnings` command-line option or by the `PYTHON_CONTEXT_AWARE_WARNINGS` environment variable.

> **備🄵**
>
> It is likely that most programs that desire thread-safe behaviour of the warnings module will also want to set the `thread_inherit_context` flag to true. That flag causes threads created by `threading.Thread` to start with a copy of the context variables from the thread starting it. When true, the context established by `catch_warnings` in one thread will also apply to new threads started by it. If false, new threads will start with an empty warnings context variable, meaning that any filtering that was established by a `catch_warnings` context manager will no longer be active.

在 3.14 版的變更: Added the `sys.flags.context_aware_warnings` flag and the use of a context variable for `catch_warnings` if the flag is true. Previous versions of Python acted as if the flag was always set to false.

29.7 `dataclasses` --- Data Classes

原始碼： Lib/dataclasses.py

This module provides a decorator and functions for automatically adding generated *special methods* such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in **PEP 557**.

The member variables to use in these generated methods are defined using **PEP 526** type annotations. For example, this code:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

will add, among other things, a `__init__()` that looks like:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class: it is not directly specified in the `InventoryItem` definition shown above.

在 3.7 版被加入.

29.7.1 模組🄵容

`@dataclasses.`**`dataclass`**`(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False, match_args=True, kw_only=False, slots=False, weakref_slot=False)`

This function is a *decorator* that is used to add generated *special methods* to classes, as described below.

The `@dataclass` decorator examines the class to find `field`s. A `field` is defined as a class variable that has a *type annotation*. With two exceptions described below, nothing in `@dataclass` examines the type specified in the variable annotation.

The order of the fields in all of the generated methods is the order in which they appear in the class definition.

如下所述，`@dataclass` 裝飾器會向類🄵新增各種"dunder" 方法。如果類🄵中已存在任何新增的方法，則行🄵會取🄵於參數，如下方文件所述。裝飾器會回傳呼叫它的同一個類🄵；不會建立新類🄵。

If `@dataclass` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `@dataclass` are equivalent:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False,
           match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

`@dataclass` 的參數是:

- *init*: 如果🄵真（預設值），將生成一個 `__init__()` 方法。

  如果該類🄵已經定義了 `__init__()`，則此參數會被忽略。

- *repr*: If true (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the

class. Fields that are marked as being excluded from the repr are not included. For example: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

如果該類🄵已經定義了 `__repr__()`，則此參數會被忽略。

- *eq*: If true (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

  如果該類🄵已經定義了 `__eq__()`，則此參數會被忽略。

- *order*: If true (the default is `False`), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If *order* is true and *eq* is false, a *ValueError* is raised.

  如果該類🄵已經定義了 `__lt__()`、`__le__()`、`__gt__()` 或 `__ge__()` 中的任何一個，則引發 *TypeError*。

- *unsafe_hash*: If true, force `dataclasses` to create a `__hash__()` method, even though it may not be safe to do so. Otherwise, generate a `__hash__()` method according to how *eq* and *frozen* are set. The default value is `False`.

  `__hash__()` is used by built-in *hash()*, and when objects are added to hashed collections such as dictionaries and sets. Having a `__hash__()` implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer's intent, the existence and behavior of `__eq__()`, and the values of the *eq* and *frozen* flags in the `@dataclass` decorator.

  By default, `@dataclass` will not implicitly add a `__hash__()` method unless it is safe to do so. Neither will it add or change an existing explicitly defined `__hash__()` method. Setting the class attribute `__hash__ = None` has a specific meaning to Python, as described in the `__hash__()` documentation.

  If `__hash__()` is not explicitly defined, or if it is set to `None`, then `@dataclass` *may* add an implicit `__hash__()` method. Although not recommended, you can force `@dataclass` to create a `__hash__()` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can still be mutated. This is a specialized use case and should be considered carefully.

  Here are the rules governing implicit creation of a `__hash__()` method. Note that you cannot both have an explicit `__hash__()` method in your dataclass and set `unsafe_hash=True`; this will result in a *TypeError*.

  如果 *eq* 和 *frozen* 都🄵真，預設情🄵下 `@dataclass` 會🄵你生成一個 `__hash__()` 方法。如果 *eq* 🄵真且 *frozen* 🄵假，`__hash__()` 將被設定🄵 `None`，🄵將其標記🄵不可雜🄵（因🄵它是可變的）。如果 *eq* 🄵假，`__hash__()` 將保持不變，這意味著將使用超類🄵的 `__hash__()` 方法（如果超類🄵是 *object*，這意味著它將回退到基於 id 的雜🄵）。

- *frozen*: If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. See the *discussion* below.

  如果類🄵中定義了 `__setattr__()` 或 `__delattr__()` 且 *frozen* 🄵真，則會引發 *TypeError*。

- *match_args*: If true (the default is `True`), the `__match_args__` tuple will be created from the list of non keyword-only parameters to the generated `__init__()` method (even if `__init__()` is not generated, see above). If false, or if `__match_args__` is already defined in the class, then `__match_args__` will not be generated.

  在 3.10 版被加入.

- *kw_only*: If true (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called. See the *parameter* glossary entry for details. Also see the *KW_ONLY* section.

  Keyword-only fields are not included in `__match_args__`.

  在 3.10 版被加入.

- *slots*: If true (the default is `False`), `__slots__` attribute will be generated and new class will be returned instead of the original one. If `__slots__` is already defined in the class, then *`TypeError`* is raised.

  > **⚠ 警告**
  >
  > Passing parameters to a base class `__init_subclass__()` when using `slots=True` will result in a *`TypeError`*. Either use `__init_subclass__` with no parameters or use default values as a workaround. See gh-91126 for full details.

  在 3.10 版被加入.

  在 3.11 版的變更: If a field name is already included in the `__slots__` of a base class, it will not be included in the generated `__slots__` to prevent overriding them. Therefore, do not use `__slots__` to retrieve the field names of a dataclass. Use *`fields()`* instead. To be able to determine inherited slots, base class `__slots__` may be any iterable, but *not* an iterator.

- *weakref_slot*: If true (the default is `False`), add a slot named "__weakref__", which is required to make an instance *`weakref-able`*. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

  在 3.11 版被加入.

`field`s may optionally specify a default value, using normal Python syntax:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

In this example, both `a` and `b` will be included in the added `__init__()` method, which will be defined as:

```
def __init__(self, a: int, b: int = 0):
```

`TypeError` will be raised if a field without a default value follows a field with a default value. This is true whether this occurs in a single class, or as a result of class inheritance.

`dataclasses.`**`field`**(*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*, *kw_only=MISSING*, *doc=None*)

For common and simple use cases, no other functionality is required. There are, however, some dataclass features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided `field()` function. For example:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

As shown above, the *`MISSING`* value is a sentinel object used to detect if some parameters are provided by the user. This sentinel is used because `None` is a valid value for some parameters with a distinct meaning. No code should directly use the *`MISSING`* value.

`field()` 的參數是:

- *default*: 如果有提供，這將是該欄位的預設值。這是必需的，因🄵 `field()` 呼叫本身會替🄵預設值的正常位置。

- *default_factory*: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both *default* and *default_factory*.
- *init*: If true (the default), this field is included as a parameter to the generated `__init__()` method.
- *repr*: If true (the default), this field is included in the string returned by the generated `__repr__()` method.
- *hash*: This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If false, this field is excluded from the generated `__hash__()`. If `None` (the default), use the value of *compare*: this would normally be the expected behavior, since a field should be included in the hash if it's used for comparisons. Setting this value to anything other than `None` is discouraged.

  One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type's hash value. Even if a field is excluded from the hash, it will still be used for comparisons.
- *compare*: If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).
- *metadata*: This can be a mapping or `None`. `None` is treated as an empty dict. This value is wrapped in *`MappingProxyType()`* to make it read-only, and exposed on the *`Field`* object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.
- *kw_only*: If true, this field will be marked as keyword-only. This is used when the generated `__init__()` method's parameters are computed.

  Keyword-only fields are also not included in `__match_args__`.

  在 3.10 版被加入.
- *doc*: optional docstring for this field.

  在 3.14 版被加入.

If the default value of a field is specified by a call to `field()`, then the class attribute for this field will be replaced by the specified *default* value. If *default* is not provided, then the class attribute will be deleted. The intent is that after the *`@dataclass`* decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute `C.z` will be `10`, the class attribute `C.t` will be `20`, and the class attributes `C.x` and `C.y` will not be set.

class dataclasses.**Field**

`Field` objects describe each defined field. These objects are created internally, and are returned by the *`fields()`* module-level method (see below). Users should never instantiate a `Field` object directly. Its documented attributes are:

- `name`：欄位的名稱。
- `type`：欄位的型🄵。
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, `metadata`, and `kw_only` have the identical meaning and values as they do in the *`field()`* function.

Other attributes may exist, but they are private and must not be inspected or relied on.

class dataclasses.**InitVar**

`InitVar[T]` type annotations describe variables that are *init-only*. Fields annotated with `InitVar` are considered pseudo-fields, and thus are neither returned by the *`fields()`* function nor used in any way except adding them as parameters to `__init__()` and an optional *`__post_init__()`*.

dataclasses.**fields**(*class_or_instance*)

Returns a tuple of *`Field`* objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises *`TypeError`* if not passed a dataclass or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

Converts the dataclass *obj* to a dict (by using the factory function *dict_factory*). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *`copy.deepcopy()`*.

Example of using `asdict()` on nested dataclasses:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

To create a shallow copy, the following workaround may be used:

```
{field.name: getattr(obj, field.name) for field in fields(obj)}
```

`asdict()` raises *`TypeError`* if *obj* is not a dataclass instance.

dataclasses.**astuple**(*obj*, *, *tuple_factory=tuple*)

Converts the dataclass *obj* to a tuple (by using the factory function *tuple_factory*). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *`copy.deepcopy()`*.

從前面的例子繼續:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

`astuple()` raises *`TypeError`* if *obj* is not a dataclass instance.

dataclasses.**make_dataclass**(*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*, *decorator=dataclass*)

Creates a new dataclass with name *cls_name*, fields as defined in *fields*, base classes as given in *bases*, and initialized with a namespace as given in *namespace*. *fields* is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, *`typing.Any`* is used for `type`. The

values of *init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots*, and *weakref_slot* have the same meaning as they do in *@dataclass*.

If *module* is defined, the `__module__` attribute of the dataclass is set to that value. By default, it is set to the module name of the caller.

The *decorator* parameter is a callable that will be used to create the dataclass. It should take the class object as a first argument and the same keyword arguments as *@dataclass*. By default, the *@dataclass* function is used.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the *@dataclass* function to convert that class to a dataclass. This function is provided as a convenience. For example:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

相當於:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

在 3.14 版被加入: Added the *decorator* parameter.

`dataclasses.`**`replace`**(*obj*, */*, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*. If *obj* is not a Data Class, raises *TypeError*. If keys in *changes* are not field names of the given dataclass, raises *TypeError*.

The newly returned object is created by calling the `__init__()` method of the dataclass. This ensures that *__post_init__()*, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to `replace()` so that they can be passed to `__init__()` and *__post_init__()*.

It is an error for *changes* to contain any fields that are defined as having `init=False`. A *ValueError* will be raised in this case.

Be forewarned about how `init=False` fields work during a call to `replace()`. They are not copied from the source object, but rather are initialized in *__post_init__()*, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

Dataclass instances are also supported by generic function *copy.replace()*.

`dataclasses.`**`is_dataclass`**(*obj*)

Return `True` if its parameter is a dataclass (including subclasses of a dataclass) or an instance of one, otherwise return `False`.

If you need to know if a class is an instance of a dataclass (and not a dataclass itself), then add a further check for `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

`dataclasses.`**`MISSING`**

A sentinel value signifying a missing default or default_factory.

dataclasses.**KW_ONLY**

A sentinel value used as a type annotation. Any fields after a pseudo-field with the type of KW_ONLY are marked as keyword-only fields. Note that a pseudo-field of type KW_ONLY is otherwise completely ignored. This includes the name of such a field. By convention, a name of _ is used for a KW_ONLY field. Keyword-only fields signify __init__() parameters that must be specified as keywords when the class is instantiated.

In this example, the fields y and z will be marked as keyword-only fields:

```
@dataclass
class Point:
    x: float
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

In a single dataclass, it is an error to specify more than one field whose type is KW_ONLY.

在 3.10 版被加入.

exception dataclasses.**FrozenInstanceError**

Raised when an implicitly defined __setattr__() or __delattr__() is called on a dataclass which was defined with frozen=True. It is a subclass of *AttributeError*.

29.7.2 Post-init processing

dataclasses.**__post_init__**()

When defined on the class, it will be called by the generated __init__(), normally as self.__post_init__(). However, if any InitVar fields are defined, they will also be passed to __post_init__() in the order they were defined in the class. If no __init__() method is generated, then __post_init__() will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

The __init__() method generated by *@dataclass* does not call base class __init__() methods. If the base class has an __init__() method that has to be called, it is common to call this method in a *__post_init__()* method:

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Note, however, that in general the dataclass-generated __init__() methods don't need to be called, since the derived dataclass will take care of initializing all fields of any base class that is a dataclass itself.

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how *`replace()`* handles `init=False` fields.

29.7.3 類🄵變數

One of the few places where *`@dataclass`* actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526**. It does this by checking if the type of the field is *`typing.ClassVar`*. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level *`fields()`* function.

29.7.4 Init-only variables

Another place where *`@dataclass`* inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type *`InitVar`*. If a field is an *`InitVar`*, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level *`fields()`* function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional *`__post_init__()`* method. They are not otherwise used by dataclasses.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, *`fields()`* will return *`Field`* objects for `i` and `j`, but not for `database`.

29.7.5 凍結實例

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the *`@dataclass`* decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a *`FrozenInstanceError`* when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__()` cannot use simple assignment to initialize fields, and must use `object.__setattr__()`.

29.7.6 繼承

When the dataclass is being created by the *`@dataclass`* decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at *`object`*) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

The final list of fields is, in order, `x`, `y`, `z`. The final type of `x` is *int*, as specified in class `C`.

The generated `__init__()` method for `C` will look like:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.7.7 Re-ordering of keyword-only parameters in `__init__()`

After the parameters needed for `__init__()` are computed, any keyword-only parameters are moved to come after all regular (non-keyword-only) parameters. This is a requirement of how keyword-only parameters are implemented in Python: they must come after non-keyword-only parameters.

In this example, `Base.y`, `Base.w`, and `D.t` are keyword-only fields, and `Base.x` and `D.z` are regular fields:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

The generated `__init__()` method for `D` will look like:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int = 0):
```

Note that the parameters have been re-ordered from how they appear in the list of fields: parameters derived from regular fields are followed by parameters derived from keyword-only fields.

The relative ordering of keyword-only parameters is maintained in the re-ordered `__init__()` parameter list.

29.7.8 預設工廠函式

If a *field()* specifies a *default_factory*, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```
mylist: list = field(default_factory=list)
```

If a field is excluded from `__init__()` (using `init=False`) and the field also specifies *default_factory*, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.

29.7.9 可變預設值

Python stores default member variable values in class attributes. Consider this example, not using dataclasses:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class `C` share the same class variable `x`, as expected.

Using dataclasses, *if* this code was valid:

```
@dataclass
class D:
    x: list = []      # 這段程式碼會引發 ValueError
    def add(self, element):
        self.x.append(element)
```

它會生成類似的程式碼:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

This has the same issue as the original example using class `C`. That is, two instances of class `D` that do not specify a value for `x` when creating a class instance will share the same copy of `x`. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, the *`@dataclass`* decorator will raise a *`ValueError`* if it detects an unhashable default parameter. The assumption is that if a value is unhashable, it is mutable. This is a partial solution, but it does protect against many common errors.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

在 3.11 版的變更: Instead of looking for and disallowing objects of type *`list`*, *`dict`*, or *`set`*, unhashable objects are now not allowed as default values. Unhashability is used to approximate mutability.

29.7.10 Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors:

- The value for the field passed to the dataclass's `__init__()` method is passed to the descriptor's `__set__()` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor's `__get__()` or `__set__()` method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, *`@dataclass`* will call the descriptor's `__get__()` method using its class access form: `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field's default. On the other hand, if the descriptor raises *`AttributeError`* in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, obj, type):
        if obj is None:
```

(繼續下一頁)

(繼續上一頁)

```
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor = IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)   # 100
i.quantity_on_hand = 2.5    # 以 2.5 呼叫 __set__
print(i.quantity_on_hand)   # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

29.8 `contextlib` --- `with` 陳述式工具程式

原始碼： Lib/contextlib.py

This module provides utilities for common tasks involving the `with` statement. For more information see also 情境管理器型🄵 and context-managers.

29.8.1 Utilities

Functions and classes provided:

class contextlib.**AbstractContextManager**

An *abstract base class* for classes that implement `object.__enter__()` and `object.__exit__()`. A default implementation for `object.__enter__()` is provided which returns `self` while `object.__exit__()` is an abstract method which by default returns `None`. See also the definition of 情境管理器型🄵.

在 3.6 版被加入.

class contextlib.**AbstractAsyncContextManager**

An *abstract base class* for classes that implement `object.__aenter__()` and `object.__aexit__()`. A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of async-context-managers.

在 3.7 版被加入.

@contextlib.**contextmanager**

This function is a *decorator* that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`.

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)
```

The function can then be used like this:

```
>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

contextmanager() uses *ContextDecorator* so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise "one-shot" context managers created by *contextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators).

在 3.2 版的變更: Use of *ContextDecorator*.

`@contextlib.`**`asynccontextmanager`**

Similar to *contextmanager()*, but creates an asynchronous context manager.

This function is a *decorator* that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an *asynchronous generator* function.

一個簡單範例:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

在 3.7 版被加入.

Context managers defined with *asynccontextmanager()* can be used either as decorators or with `async with` statements:

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... async code ...
```

When used as a decorator, a new generator instance is implicitly created on each function call. This allows the otherwise "one-shot" context managers created by *asynccontextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators.

在 3.10 版的變更: Async context managers created with *asynccontextmanager()* can be used as decorators.

contextlib.**closing**(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

備🄵

Most types managing resources support the *context manager* protocol, which closes *thing* on leaving the `with` statement. As such, `closing()` is most useful for third party types that don't support context managers. This example is purely for illustration purposes, as *urlopen()* would normally be used in a context manager.

contextlib.**aclosing**(*thing*)

Return an async context manager that calls the `aclose()` method of *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import asynccontextmanager
```

(繼續下一頁)

(繼續上一頁)

```
@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by `break` or an exception. For example:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator's async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn't run after the lifetime of some task it depends on).

在 3.10 版被加入.

`contextlib.`**`nullcontext`**(*enter_result=None*)

Return a context manager that returns *enter_result* from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

一個使用 *enter_result* 的範例：

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

It can also be used as a stand-in for asynchronous context managers:

```
async def send_http(session=None):
    if not session:
        # If no http session, create it with aiohttp
        cm = aiohttp.ClientSession()
    else:
        # Caller is responsible for closing the session
        cm = nullcontext(session)

    async with cm as session:
        # Send http requests with session
```

在 3.7 版被加入.

在 3.10 版的變更: *asynchronous context manager* support was added.

contextlib.**suppress**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a `with` statement and then resumes execution with the first statement following the end of the `with` statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

舉例來🄵:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is *reentrant*.

If the code within the `with` block raises a *BaseExceptionGroup*, suppressed exceptions are removed from the group. Any exceptions of the group which are not suppressed are re-raised in a new group which is created using the original group's *derive()* method.

在 3.4 版被加入.

在 3.12 版的變更: `suppress` now supports suppressing exceptions raised as part of a *BaseExceptionGroup*.

contextlib.**redirect_stdout**(*new_target*)

Context manager for temporarily redirecting *sys.stdout* to another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of *help()* normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an *io.StringIO* object. The replacement stream is returned from the `__enter__` method and so is available as the target of the `with` statement:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

To send the output of *help()* to a file on disk, redirect the output to a regular file:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of *help()* to *sys.stderr*:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on *sys.stdout* means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is *reentrant*.

在 3.4 版被加入.

contextlib.**redirect_stderr**(*new_target*)

Similar to *redirect_stdout()* but redirecting *sys.stderr* to another file or file-like object.

This context manager is *reentrant*.

在 3.5 版被加入.

contextlib.**chdir**(*path*)

Non parallel-safe context manager to change the current working directory. As this changes a global state, the working directory, it is not suitable for use in most threaded or async contexts. It is also not suitable for most non-linear code execution, like generators, where the program execution is temporarily relinquished -- unless explicitly desired, you should not yield when this context manager is active.

This is a simple wrapper around *chdir()*, it changes the current working directory upon entering and restores the old one on exit.

This context manager is *reentrant*.

在 3.11 版被加入.

class contextlib.**ContextDecorator**

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by *contextmanager()*, so you get this functionality automatically.

`ContextDecorator` 範例:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
```

(繼續下一頁)

(繼續上一頁)

```
...
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

備F

As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

在 3.2 版被加入.

class contextlib.**AsyncContextDecorator**

Similar to *ContextDecorator* but only for asynchronous functions.

`AsyncContextDecorator` 範例:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...     async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

在 3.10 版被加入.

class contextlib.**ExitStack**

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

The `__enter__()` method returns the *ExitStack* instance, and performs no additional operations.

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

在 3.3 版被加入.

enter_context(*cm*)

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

在 3.11 版的變更: Raises *TypeError* instead of *AttributeError* if *cm* is not a context manager.

push(*exit*)

Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, */*, **args*, ***kwds*)

Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()

Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()

Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.**AsyncExitStack**

An asynchronous context manager, similar to *ExitStack*, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The *close()* method is not implemented; *aclose()* must be used instead.

async enter_async_context(*cm*)

Similar to *ExitStack.enter_context()* but expects an asynchronous context manager.

在 3.11 版的變更: Raises *TypeError* instead of *AttributeError* if *cm* is not an asynchronous context manager.

push_async_exit(*exit*)

Similar to *ExitStack.push()* but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, */*, **args*, ***kwds*)

Similar to *ExitStack.callback()* but expects a coroutine function.

async aclose()

Similar to *ExitStack.close()* but properly handles awaitables.

Continuing the example for *asynccontextmanager()*:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

在 3.7 版被加入.

29.8.2 Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use `with` statements to manage arbitrary resources that don't natively support the context management protocol.

Catching exceptions from `__enter__` methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the `with` statement body or the context manager's `__exit__` method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with `try`/`except`/`finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a `with` statement.

Cleaning up in an `__enter__` implementation

As noted in the documentation of *ExitStack.push()*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *ExitStack.callback()* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

也參考

PEP 343 - "with" 陳述式
The specification, background, and examples for the Python `with` statement.

29.8.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *`contextmanager()`* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()*, *redirect_stdout()*, and *chdir()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
```

(繼續下一頁)

(繼續上一頁)

```
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.9 abc --- 抽象基底類🄵

原始碼： Lib/abc.py

如同在 **PEP 3119** 中所述，該模組提供了在 Python 中定義抽象基底類🄵 (ABC) 的基礎元件；若想🄵解🄵什🄵需要在 Python 中增加這個模組，請見 PEP 文件。（也請見 **PEP 3141** 以及 *numbers* 模組以🄵解基於 ABC 的數字型🄵階層關🄵。）

collections 模組中有一些衍生自 ABC 的具體類🄵；當然這些類🄵還可以進一步衍生出其他類🄵。此外，*collections.abc* 子模組中有一些 ABC 可被用於測試一個類🄵或實例是否提供特定介面，例如它是否可雜🄵 *(hashable)* 或它是否🄵對映。

該模組提供了一個用來定義 ABC 的元類🄵 (metaclass) *ABCMeta* 和另一個以繼承的方式定義 ABC 的工具類🄵*ABC*：

class abc.**ABC**

一個使用*ABCMeta* 作🄵元類🄵的工具類🄵。抽象基底類🄵可以透過自 ABC 衍生而建立，這就避免了在某些情🄵下會令人混淆的元類🄵用法，用法如以下範例：

```
from abc import ABC

class MyABC(ABC):
    pass
```

注意 ABC 的型🄵仍然是*ABCMeta*，因此繼承 ABC 仍然需要關注使用元類🄵的注意事項，如多重繼承可能會導致元類🄵衝突。當然你也可以傳入元類🄵關鍵字🄵直接使用 ABCMeta 來定義一個抽象基底類🄵，例如：

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

在 3.4 版被加入.

class abc.**ABCMeta**

用於定義抽象基底類🄵（ABC）的元類🄴。

使用該元類🄵以建立一個 ABC。一個 ABC 可以像 mix-in 類🄵一樣直接被子類🄵繼承。你也可以將不相關的具體類🄵（甚至是🄵建類🄵）和 ABC 🄵🄵🄵「🄵擬子類🄵 (virtual subclass)」——這些類🄵以及它們的子類🄵會被🄵建函式*issubclass()* 識🄵🄵已🄵🄵 ABC 的子類🄴，但是該 ABC 不會出現在其 MRO（Method Resolution Order，方法解析順序）中，由該 ABC 所定義的方法實作也不可呼叫（即使透過*super()* 呼叫也不行）。[1]

使用 ABCMeta 作🄵元類🄵建立的類🄵含有以下的方法：

register(*subclass*)

將子類🄵🄵🄵🄵該 ABC 的「抽象子類🄵」，例如：

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

在 3.3 版的變更: 回傳已🄵🄵的子類🄵，使其能🄵作🄵類🄵裝飾器。

在 3.4 版的變更: 你可以使用*get_cache_token()* 函式來檢測對 register() 的呼叫。

你也可以覆寫 (override) 🄵擬基底類🄵中的這個方法：

__subclasshook__(*subclass*)

（必須定義🄵類🄵方法。）

檢查 *subclass* 是否該被認🄴是該 ABC 的子類🄴，也就是🄵你可以直接自訂*issubclass()* 的行🄵，而不用對於那些你希望定義🄵該 ABC 的子類🄵的類🄵都個🄵呼叫*register()* 方法。（這個類🄵方法是在 ABC 的 __subclasscheck__() 方法中呼叫。）

此方法必須回傳 True、False 或是*NotImplemented*。如果回傳 True，*subclass* 就會被認🄵是這個 ABC 的子類🄵。如果回傳 False，*subclass* 就會被判定🄵非該 ABC 的子類🄵，即便正常情🄵應如此。如果回傳 NotImplemented，子類🄵檢查會按照正常機制繼續執行。

🄵了對這些概念做一演示，請見以下定義 ABC 的範例：

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
```

(繼續下一頁)

[1] C++ 程式設計師需要注意到 Python 中🄵擬基底類🄵的概念和 C++ 中的🄵不相同。

(繼續上一頁)

```
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC `MyIterable` 定義了作🄵抽象方法的一個標准🄵代方法 `__iter__()`。這🄵給定的實作仍可在子類🄵中被呼叫。`get_iterator()` 方法也是 `MyIterable` 抽象基底類🄵的一部分，但它不必被非抽象衍生類🄵覆寫。

這🄵定義的*`__subclasshook__()`* 類🄵方法🄵明任何在其 `__dict__` (或在其透過 `__mro__` 列表存取的基底類🄵) 中具有 `__iter__()` 方法的類🄵也都會被視🄵 `MyIterable`。

最後，即使 `Foo` 🄵有定義 `__iter__()` 方法（它使用了以 `__len__()` 和 `__getitem__()` 所定義的舊式可🄵代物件協定），最末一行使其成🄵 `MyIterable` 的一個🄵擬子類🄵。請注意這不會使 `get_iterator` 成🄵 `Foo` 的一個可用方法，所以它是需要被另外提供的。

`abc` 模組也提供了這些裝飾器：

`@abc.`**`abstractmethod`**

用於表示抽象方法的裝飾器。

類🄵的元類🄵是*`ABCMeta`* 或是從該類🄵衍生才能使用此裝飾器。一個具有衍生自 `ABCMeta` 之元類🄵的類🄵不可以被實例化，除非它全部的抽象方法和特性均已被覆寫。抽象方法可透過任何一般的'super' 呼叫機制來呼叫。`abstractmethod()` 可被用於🄵特性和描述器宣告的抽象方法。

僅在使用*`update_abstractmethods()`* 函式時，才能🄵動態地🄵一個類🄵新增抽象方法，或者嘗試在方法或類🄵被建立後修改其抽象狀態。`abstractmethod()` 只會影響使用常規繼承所衍生出的子類🄵；透過 ABC 的*`register()`* 方法🄵🄵的「🄵擬子類🄵」不會受到影響。

當 `abstractmethod()` 與其他方法描述器 (method descriptor) 配合應用時，它應被當最🄵層的裝飾器，如以下用法範例所示：

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
```

(繼續下一頁)

(繼續上一頁)

```
        ...
    x = property(_get_x, _set_x)
```

🄵了能正確地與 ABC 機制實作相互操作，描述器必須使用 `__isabstractmethod__` 將自身標識🄵抽象的。一般來🄵，如果被用於組成描述器的任一方法是抽象的，則此屬性應當🄵 `True`。例如，Python 的🄵建*property* 所做的就等價於：

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

備🄵

不同於 Java 抽象方法，這些抽象方法可能具有一個實作。這個實作可在覆寫它的類🄵上透過*super()* 機制來呼叫。這在使用協作多重繼承 (cooperative multiple-inheritance) 的框架中，可以被用作 super 呼叫的一個端點 (end-point)。

`abc` 模組還支援下列舊式裝飾器：

@abc.abstractclassmethod

在 3.2 版被加入.

在 3.3 版之後被🄵用: 現在可以讓*classmethod* 配合*abstractmethod()* 使用，使得此裝飾器變得冗余。

🄵建*classmethod()* 的子類🄵，表示🄵一個抽象類🄵方法。在其他方面它都類似於*abstractmethod()*。

這個特例已被🄵用，因🄵現在當*classmethod()* 裝飾器應用於抽象方法時已會被正確地標識🄵是抽象的：

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.abstractstaticmethod

在 3.2 版被加入.

在 3.3 版之後被🄵用: 現在可以讓*staticmethod* 配合*abstractmethod()* 使用，使得此裝飾器變得冗余。

🄵建*staticmethod()* 的子類🄵，表示🄵一個抽象🄵態方法。在其他方面它都類似於*abstractmethod()*。

這個特例已被🄵用，因🄵現在當*staticmethod()* 裝飾器應用於抽象方法時已會被正確地標識🄵是抽象的：

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.**abstractproperty**

在 3.3 版之後被🄵用：現在可以讓*property*、*property.getter()*、*property.setter()*和*property.deleter()*配合*abstractmethod()*使用，使得此裝飾器變得冗余。

🄵建*property()*的子類🄵，表示🄵一個抽象特性。

這個特例已被🄵用，因🄵現在當*property()*裝飾器應用於抽象方法時已會被正確地標識🄵是抽象的：

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

上面的例子定義了一個唯讀特性；你也可以透過適當地將一個或多個底層方法標記🄵抽象的來定義可讀寫的抽象特性：

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

如果只有某些元件是抽象的，則只需更新那些元件即可在子類🄵中建立具體的特性：

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc 模組也提供了這些函式：

abc.**get_cache_token**()

回傳目前 ABC 快取令牌 (cache token)。

此令牌是一個（支援相等性測試的）不透明物件 (opaque object)，用於🄵🄵擬子類🄵標識抽象基底類🄵快取的目前版本。此令牌會在任何 ABC 上每次呼叫*ABCMeta.register()*時發生更改。

在 3.4 版被加入.

abc.**update_abstractmethods**(*cls*)

重新計算一個抽象類🄵之抽象狀態的函式。如果一個類🄵的抽象方法在建立後被實作或被修改，則應當呼叫此函式。通常此函式應在一個類🄵裝飾器🄵部被呼叫。

回傳 *cls*，使其能🄵用作🄵類🄵的裝飾器。

如果 *cls* 不是*ABCMeta*的實例則不做任何操作。

備🄵

此函式會假定 *cls* 的超類🄵 (superclass) 已經被更新。它不會更新任何子類🄵。

在 3.10 版被加入.

29.10 atexit --- 退出處理函式

atexit 模組定義了🄵🄵和🄵銷清理函式 (cleanup function) 的函式，這樣的🄵🄵函式會在直譯器正常終止時自動執行。*atexit* 按照這些函式🄵🄵之反序來運行這些函式；如果你有🄵🄵 A、B 和 C，則在直譯器終止時它們將按 C、B、A 的順序運行。

注意：當程式被一個不是來自 Python 的訊號終止、偵測到有 Python 嚴重🄵部錯誤時或者*os._exit()* 被呼叫時，透過此模組🄵🄵的函式就不會被呼叫。

注意：在清理函式中🄵🄵或🄵銷函式的作用🄵未定義。

在 3.7 版的變更: 當與 C-API 子直譯器 (subinterpreter) 一起使用時，已🄵🄵函式對於它們所🄵🄵的直譯器來🄵是區域的 (local)。

atexit.**register**(*func*, **args*, ***kwargs*)

將 *func* 🄵🄵🄵要在終止時執行的函式。任何要傳遞給 *func* 的可選引數都必須作🄵引數傳遞給*register()*。可以多次🄵🄵相同的函式和引數。

在程式正常終止時（例如呼叫*sys.exit()* 或主要模組執行完成），所有已🄵🄵函式都會依照後進先出的順序呼叫。這邊做的假設是較低階的模組通常會在較高階模組之前被引入，因此較低階模組必須在比較後面才被清理。

如果在執行退出處理函式期間引發例外，則會列印回溯 (traceback)（除非引發*SystemExit*）🄵儲存例外資訊。在所有退出處理函式都有嘗試運作過後，將重新引發最後一個引發的例外。

該函式回傳 *func*，這使得可以將其作🄵裝飾器使用。

> **警告**
>
> 🄵動新執行緒或從已🄵🄵函式呼叫*os.fork()* 可能會導致 Python 主要 runtime 執行緒釋放執行緒狀態，而🄵部*threading* 例程或新行程嘗試使用該狀態所形成的競態條件 (race condition)。這可能會導致崩潰 (crash) 而不是完整關閉中止。

在 3.12 版的變更: 嘗試在已🄵🄵函式中🄵動新執行緒或*os.fork()* 新行程現在會導致*RuntimeError*。

atexit.**unregister**(*func*)

從在直譯器中止時會執行之函式串列中🄵除 *func*。如果 *func* 先前未被🄵🄵，則*unregister()* 不會執行任何操作。如果 *func* 已被🄵🄵多次，則該函式在*atexit* 呼叫堆🄵 (call stack) 中的所有存在都會被🄵除。會在🄵銷期間於🄵部使用相等性比較 (==)，因此函式參照不需要具有匹配的識🄵性。

> **也參考**
>
> ***readline* 模組**
> *atexit* 用於讀取和寫入*readline* 歷史檔案的一個不錯的範例。

29.10.1 atexit 范例

以下的簡單範例示範了模組如何在被引入時以檔案來初始化計數器，🄵在程式終止時自動儲存計數器的更新值，而不需要仰賴應用程式在終止時明確呼叫該模組。

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0
```

(繼續下一頁)

(繼續上一頁)

```
def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

位置引數和關鍵字引數也可以被傳遞給*register()*，以便在呼叫時也傳遞給已🄵🄵函式：

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit

atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

作🄵*裝飾器*使用：

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

這只適用於可以不帶引數呼叫的函式。

29.11 `traceback` --- 列印或取得堆🄵回溯 (stack traceback)

原始碼： Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It is more flexible than the interpreter's default traceback display, and therefore makes it possible to configure certain aspects of the output. Finally, it contains a utility for capturing enough information about an exception to print it later, without the need to save a reference to the actual exception. Since exceptions can be the roots of large objects graph, this utility can significantly improve memory management.

The module uses traceback objects --- these are objects of type *types.TracebackType*, which are assigned to the *__traceback__* field of *BaseException* instances.

也參考

***faulthandler* 模組**
: Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

***pdb* 模組**
: Interactive source code debugger for Python programs.

The module's API can be divided into two parts:

- Module-level functions offering basic functionality, which are useful for interactive inspection of exceptions and tracebacks.

- *TracebackException* class and its helper classes *StackSummary* and *FrameSummary*. These offer both more flexibility in the output generated and the ability to store the information necessary for later formatting without holding references to actual exception and traceback objects.

在 3.13 版被加入: Output is colorized by default and can be controlled using environment variables.

29.11.1 Module-Level Functions

traceback.**print_tb**(*tb*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to *sys.stderr*; otherwise it should be an open *file* or *file-like object* to receive the output.

> **備🄵**
>
> The meaning of the *limit* parameter is different than the meaning of *sys.tracebacklimit*. A negative *limit* value corresponds to a positive value of `sys.tracebacklimit`, whereas the behaviour of a positive *limit* value cannot be achieved with `sys.tracebacklimit`.

在 3.5 版的變更: 新增負數 *limit* 的支援。

traceback.**print_exception**(*exc*, */*, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from *print_tb()* in the following ways:

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`
- it prints the exception type and *value* after the stack trace
- if *type(value)* is *SyntaxError* and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

The optional *limit* argument has the same meaning as for *print_tb()*. If *chain* is true (the default), then chained exceptions (the *__cause__* or *__context__* attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

在 3.5 版的變更: The *etype* argument is ignored and inferred from the type of *value*.

在 3.10 版的變更: The *etype* parameter has been renamed to *exc* and is now positional-only.

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.exception(), limit=limit, file=file, chain=chain)`.

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.last_exc, limit=limit, file=file, chain=chain)`. In general it will work only after an exception has reached an interactive prompt (see *sys.last_exc*).

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for *print_tb()*.

在 3.5 版的變更: 新增負數 *limit* 的支援。

traceback.**extract_tb**(*tb*, *limit=None*)

Return a *StackSummary* object representing a list of "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for *print_tb()*. A "pre-processed" stack trace entry is a *FrameSummary* object containing attributes *filename*, *lineno*, *name*, and *line* representing the information that is usually printed for a stack trace.

traceback.**extract_stack**(*f=None*, *limit=None*)

Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

traceback.**print_list**(*extracted_list*, *file=None*)

Print the list of tuples as returned by *extract_tb()* or *extract_stack()* as a formatted stack trace to the given file. If *file* is `None`, the output is written to *sys.stderr*.

traceback.**format_list**(*extracted_list*)

Given a list of tuples or *FrameSummary* objects as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.**format_exception_only**(*exc*, */*, [*value*,]*, *show_group=False*)

Format the exception part of a traceback using an exception value such as given by *sys.last_value*. The return value is a list of strings, each ending in a newline. The list contains the exception's message, which is normally a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. Following the message, the list contains the exception's *notes*.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is `True`, and the exception is an instance of *BaseExceptionGroup*, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

在 3.10 版的變更: The *etype* parameter has been renamed to *exc* and is now positional-only.

在 3.11 版的變更: The returned list now includes any *notes* attached to the exception.

在 3.13 版的變更: *show_group* parameter was added.

traceback.**format_exception**(*exc*, */*, [*value*, *tb*,]*limit=None*, *chain=True*)

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

在 3.5 版的變更: The *etype* argument is ignored and inferred from the type of *value*.

在 3.10 版的變更: This function's behavior and signature were modified to match *print_exception()*.

traceback.**format_exc**(*limit=None*, *chain=True*)

This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)

`format_list(extract_tb(tb, limit))` 的簡寫。

traceback.**format_stack**(*f=None*, *limit=None*)

`format_list(extract_stack(f, limit))` 的簡寫。

traceback.**clear_frames**(*tb*)

Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

在 3.4 版被加入.

traceback.**walk_stack**(*f*)

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with *StackSummary.extract()*.

在 3.5 版被加入.

在 3.14 版的變更: This function previously returned a generator that would walk the stack when first iterated over. The generator returned now is the state of the stack when `walk_stack` is called.

traceback.**walk_tb**(*tb*)

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

在 3.5 版被加入.

29.11.2 `TracebackException` 物件

在 3.5 版被加入.

`TracebackException` objects are created from actual exceptions to capture data for later printing. They offer a more lightweight method of storing this information by avoiding holding references to traceback and frame objects. In addition, they expose more options to configure the output compared to the module-level functions described above.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

Capture an exception for later rendering. The meaning of *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

If *compact* is true, only data that is required by `TracebackException`'s *format()* method is saved in the class attributes. In particular, the *__context__* field is calculated only if *__cause__* is `None` and *__suppress_context__* is false.

Note that when locals are captured, they are also shown in the traceback.

max_group_width and *max_group_depth* control the formatting of exception groups (see *BaseExceptionGroup*). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

在 3.10 版的變更: 新增 *compact* 參數。

在 3.11 版的變更: 新增 *max_group_width* 和 *max_group_depth* 參數。

__cause__

A `TracebackException` of the original *__cause__*.

__context__

A `TracebackException` of the original *__context__*.

exceptions

If `self` represents an *ExceptionGroup*, this field holds a list of `TracebackException` instances representing the nested exceptions. Otherwise it is `None`.

在 3.11 版被加入.

__suppress_context__

The *__suppress_context__* value from the original exception.

__notes__

The *__notes__* value from the original exception, or `None` if the exception does not have any notes. If it is not `None` is it formatted in the traceback after the exception string.

在 3.11 版被加入.

stack

A *StackSummary* representing the traceback.

exc_type

The class of the original traceback.

在 3.13 版之後被🄵用.

exc_type_str

String display of the class of the original exception.

在 3.13 版被加入.

filename

For syntax errors - the file name where the error occurred.

lineno

For syntax errors - the line number where the error occurred.

end_lineno

For syntax errors - the end line number where the error occurred. Can be `None` if not present.

在 3.10 版被加入.

text

For syntax errors - the text where the error occurred.

offset

For syntax errors - the offset into the text where the error occurred.

end_offset

For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

在 3.10 版被加入.

msg

For syntax errors - the compiler error message.

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

print(*, *file=None*, *chain=True*)

Print to *file* (default `sys.stderr`) the exception information returned by *format()*.

在 3.11 版被加入.

format(*, *chain=True*)

Format the exception.

If *chain* is not `True`, *__cause__* and *__context__* will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *print_exception()* is a wrapper around this method which just prints the lines to a file.

format_exception_only(*, *show_group=False*)

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

When *show_group* is `False`, the generator emits the exception's message followed by its notes (if it has any). The exception message is normally a single string; however, for *SyntaxError* exceptions, it consists of several lines that (when printed) display detailed information about where the syntax error occurred.

When *show_group* is `True`, and the exception is an instance of *BaseExceptionGroup*, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

在 3.11 版的變更: The exception's *notes* are now included in the output.

在 3.13 版的變更: 新增 *show_group* 參數。

29.11.3 StackSummary 物件

在 3.5 版被加入.

`StackSummary` objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construct a `StackSummary` object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the `StackSummary` cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each `FrameSummary` are captured as object representations.

在 3.12 版的變更: Exceptions raised from *repr()* on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

classmethod from_list(*a_list*)

Construct a `StackSummary` object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with *filename*, *lineno*, *name*, *line* as the elements.

format()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

在 3.6 版的變更: Long sequences of repeated frames are now abbreviated.

format_frame_summary(*frame_summary*)

Returns a string for printing one of the frames involved in the stack. This method is called for each *FrameSummary* object to be printed by *StackSummary.format()*. If it returns `None`, the frame is omitted from the output.

在 3.11 版被加入.

29.11.4 FrameSummary 物件

在 3.5 版被加入.

A `FrameSummary` object represents a single frame in a traceback.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *, *lookup_line=True*, *locals=None*, *line=None*, *end_lineno=None*, *colno=None*, *end_colno=None*)

Represents a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frame's locals included in it. If *lookup_line* is `False`, the source code is not looked up until the `FrameSummary` has the *line* attribute accessed (which also happens when casting it to a *tuple*). *line* may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable mapping, and if supplied the variable representations are stored in the summary for later display.

`FrameSummary` instances have the following attributes:

filename

The filename of the source code for this frame. Equivalent to accessing `f.f_code.co_filename` on a frame object *f*.

lineno

The line number of the source code for this frame.

name

Equivalent to accessing `f.f_code.co_name` on a frame object *f*.

line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

end_lineno

The last line number of the source code for this frame. By default, it is set to `lineno` and indexation starts from 1.

在 3.13 版的變更: The default value changed from `None` to `lineno`.

colno

The column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

end_colno

The last column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

29.11.5 Examples of Using the Module-Level Functions

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError as exc:
```

(繼續下一頁)

(繼續上一頁)

```
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n    ~~~~~~
↪~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return tuple()[0]\n    ␣
↪      ~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_life>]
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n    ~~~~~~
↪~~~~~~~~~~~~~~^^\n',
```

(繼續下一頁)

(繼續上一頁)

```
'  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return tuple()[0]\n    ␣
↪      ~~~~~~~^^^\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(an_error)
['IndexError: tuple index out of range\n']
```

29.11.6 `TracebackException` 的使用范例

With the helper class, we have more options:

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
```

(繼續下一頁)

(繼續上一頁)

```
...     try:
...         lumberjack()
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # limit works as with the module-level functions
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

>>> # capture_locals adds local variables in frames
>>> TracebackException.from_exception(exc, limit=-2, capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # The *chain* kwarg to print() controls whether chained
>>> # exceptions are displayed
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>> TracebackException.from_exception(chained_exc).print(chain=False)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero
```

29.12 `__future__` --- Future 陳述式定義

原始碼： Lib/__future__.py

`from __future__ import feature` 形式的引入被稱🄵 future 陳述式。這些是 Python 編譯器的特殊情🄵，允許在該功能成🄵標准版本之前在包含 future 陳述式的模組中使用新的 Python 功能。

雖然這些 future 陳述式被 Python 編譯器賦予了額外的特殊意義，但它們仍然像任何其他 import 陳述式一樣執行，且*__future__* 由引入系統以和任何其他 Python 模組相同的方式處理。這個設計有三個目的：

- 🄵了避免混淆分析引入陳述式🄵預期要找到它們正在引入之模組的現有工具。
- 記🄵何時出現不相容的變更，以及何時開始🄵制執行這些變更。這是一種可執行文件的形式，可以透過引入*__future__* 🄵檢查其🄵容以程式化的方式進行檢查。
- 確保 future 陳述式在 Python 2.1 之前的版本中運行至少會🄵生 runtime 例外（*__future__* 的引入將會失敗，因🄵 2.1 之前🄵有該名稱的模組）。

29.12.1 模組🄵容

不會從*__future__* 中🄵除任何功能描述。自從在 Python 2.1 中引入以來，以下功能已透過這種機制引入到該語言中：

功能	可選的版本	🄵制性的版本	影響
`__future__.`**`nested_scopes`**	2.1.0b1	2.2	**PEP 227**: 🄵態巢狀作用域 *(Statically Nested Scopes)*
`__future__.`**`generators`**	2.2.0a1	2.3	**PEP 255**: 簡單🄵生器 *(Simple Generators)*
`__future__.`**`division`**	2.2.0a2	3.0	**PEP 238**: 更改除法運算子 *(Changing the Division Operator)*
`__future__.`**`absolute_import`**	2.5.0a1	3.0	**PEP 328**: 引入: 多列與🄵對/相對 *(Imports: Multi-Line and Absolute/Relative)*
`__future__.`**`with_statement`**	2.5.0a1	2.6	**PEP 343**: *"with"* 陳述式 *(The "with" Statement)*
`__future__.`**`print_function`**	2.6.0a2	3.0	**PEP 3105**: 使 *print* 成🄵一個函式 *(Make print a function)*
`__future__.`**`unicode_literals`**	2.6.0a2	3.0	**PEP 3112**: *Python 3000* 中的位元組字面值 *(Bytes literals in Python 3000)*
`__future__.`**`generator_stop`**	3.5.0b1	3.7	**PEP 479**: 🄵生器🄵部的 *StopIteration* 處理 *(StopIteration handling inside generators)*
`__future__.`**`annotations`**	3.7.0b1	從未[1]	**PEP 563**: *Postponed evaluation of annotations*, **PEP 649**: *Deferred evaluation of annotations using descriptors*

class `__future__.`**`_Feature`**

`__future__.py` 中的每個陳述式的形式如下:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

通常，*OptionalRelease* 會小於 *MandatoryRelease*，且兩者都是與`sys.version_info` 形式相同的 5 元組 (5-tuple):

```
(PY_MAJOR_VERSION, # 2.1.0a3 🄵面的 2; 🄵一整數
 PY_MINOR_VERSION, # 1; 🄵一整數
 PY_MICRO_VERSION, # 0; 🄵一整數
 PY_RELEASE_LEVEL, # "alpha"、"beta"、"candidate" 或 "final"; ; 🄵一字串
 PY_RELEASE_SERIAL # 3; 🄵一整數
)
```

[1] `from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the change was delayed and ultimately canceled. This feature will eventually be deprecated and removed. See **PEP 649** and **PEP 749**.

_Feature.**getOptionalRelease**()

OptionalRelease 記🄴該功能首次發布時的 Python 版本。

_Feature.**getMandatoryRelease**()

如果 *MandatoryRelease* 尚未發布，*MandatoryRelease* 會預測該功能將成🄴該語言一部分的版本。

否則 *MandatoryRelease* 會記🄴該功能是何時成🄴語言的一部分；在該版本或之後的版本中，模組不再需要 future 聲明來使用相關功能，但可以繼續使用此種引入方式。

MandatoryRelease 也可能是 `None`，這意味著計劃中的功能被🄵🄵或者仍未🄵定。

_Feature.**compiler_flag**

CompilerFlag 是（位元欄位 (bitfield)）旗標，應在第四個引數中傳遞給🄵建函式 *compile()* 以在動態編譯的程式碼中🄵用該功能。此旗標儲存在 *_Feature* 實例上的 *_Feature.compiler_flag* 屬性中。

也參考

future
編譯器如何處理 future 引入。

PEP 236 - 回到 __future__
__future__ 機制的原始提案。

29.13 gc --- 垃圾回收器介面 (Garbage Collector interface)

此 module（模組）提供可選的垃圾回收器介面，提供的功能包括：關閉回收器、調整回收頻率、設定除錯選項。它同時提供對回收器有找到但是無法釋放的不可達物件 (unreachable object) 的存取。由於 Python 使用了帶有參照計數的回收器，如果你確定你的程式不會🄴生參照🄵圈 (reference cycle)，你可以關閉回收器。可以透過呼叫 `gc.disable()` 關閉自動垃圾回收。若要🄴一個存在記憶體流失的程式 (leaking program) 除錯，請呼叫 `gc.set_debug(gc.DEBUG_LEAK)`；需要注意的是，它包含 `gc.DEBUG_SAVEALL`，使得被回收的物件會被存放在 gc.garbage 中以待檢查。

gc module 提供下列函式：

gc.**enable**()

🄴用自動垃圾回收。

gc.**disable**()

停用自動垃圾回收。

gc.**isenabled**()

如果🄴用了自動回收則回傳 `True`。

gc.**collect**(*generation=2*)

🄴動垃圾回收。可選的引數 *generation* 可以是一個指明需要回收哪一代垃圾的整數（從 0 到 2）。當 generation 數值無效時，會引發 *ValueError* 例外。可回收物件與不可回收物件的總和會被回傳。

Calling `gc.collect(0)` will perform a GC collection on the young generation.

Calling `gc.collect(1)` will perform a GC collection on the young generation and an increment of the old generation.

Calling `gc.collect(2)` or `gc.collect()` performs a full collection

每當執行完整回收或最高代 (2) 回收時，🄵多個🄵建型🄵所維護的空🄵列表會被清空。🄵了特定型🄵的實現，特🄵是 *float*，在某些空🄵列表中🄵非所有項目都會被釋放。

當直譯器已經執行收集時呼叫 `gc.collect()` 的效果是未定義的。

在 3.14 版的變更: `generation=1` performs an increment of collection.

gc.**set_debug**(*flags*)

設定垃圾回收器的除錯旗標。除錯資訊會被寫入 `sys.stderr`。請見下方的除錯旗標列表，可以使用位元操作 (bit operation) 進行設定以控制除錯程式。

gc.**get_debug**()

回傳目前設置的除錯旗標。

gc.**get_objects**(*generation=None*)

回傳一個包含回收器正在追🄵的所有物件的 list，除去所回傳的 list。如果 *generation* 不🄵 `None`，只回傳以下物件。

- 0: 年輕代中的所有物件。
- 1: 🄵有物件，因🄵不存在第 1 代（截至 Python 3.14）
- 2: 年老代中的所有物件。

在 3.8 版的變更: 新增 *generation* 參數。

在 3.14 版的變更: Generation 1 is removed

引發一個附帶引數 `generation` 的稽核事件 *(auditing event)* `gc.get_objects`。

gc.**get_stats**()

回傳一個包含三個字典物件的 list，每個字典分🄵包含對應代中自從直譯器開始執行後的垃圾回收統計資料。字典的鍵的數目在將來可能會改變，但目前每個字典包含以下項目：

- `collections` 是該代被回收的次數；
- `collected` 是該代中被回收的物件總數；
- `uncollectable` 是在這一代中被發現無法回收的物件總數（因此被移到*garbage* list 中）。

在 3.4 版被加入.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])

設定垃圾回收🄵值（回收頻率）。將 *threshold0* 設🄵零會停止回收。

The GC classifies objects into two generations depending on whether they have survived a collection. New objects are placed in the young generation. If an object survives a collection it is moved into the old generation.

In order to decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. For each collection, all the objects in the young generation and some fraction of the old generation is collected.

In the free-threaded build, the increase in process memory usage is also checked before running the collector. If the memory usage has not increased by 10% since the last collection and the net number of object allocations has not exceeded 40 times *threshold0*, the collection is not run.

The fraction of the old generation that is collected is **inversely** proportional to *threshold1*. The larger *threshold1* is, the slower objects in the old generation are collected. For the default value of 10, 1% of the old generation is scanned during each collection.

threshold2 is ignored.

See Garbage collector design for more information.

在 3.14 版的變更: *threshold2* is ignored

gc.**get_count**()

將目前回收計數以 `(count0, count1, count2)` 形式的 tuple 回傳。

gc.**get_threshold**()

將目前回收🄵值以 `(threshold0, threshold1, threshold2)` 形式的 tuple 回傳。

gc.**get_referrers**(**objs*)

回傳包含直接參照 objs 中任一個物件的物件 list。這個函式只定位支援垃圾回收的容器；參照了其它物件但不支援垃圾回收的擴充套件型🄵無法被找到。

需要注意的是，已經解除參照的物件，但仍存在於參照🄵圈中未被回收時，該物件仍然會被作🄵參照者出現在回傳的 list 中。若只要取得目前正在參照的物件，需要在呼叫*get_referrers()* 之前呼叫*collect()*。

> **⚠ 警告**
>
> 在使用*get_referrers()* 回傳的物件時必須要小心，因🄵其中的一些物件可能仍在建構中而處於暫時無效的狀態。不要把*get_referrers()* 用於除錯以外的其它目的。

引發一個附帶引數 `objs` 的稽核事件 `gc.get_referrers`。

gc.**get_referents**(**objs*)

回傳包含被任意一個引數直接參照之物件的 list。回傳的被參照物件是有被引數的 C 語言級🄵 `tp_traverse` 方法（若存在）🄵覽到的物件，可能不是所有的實際直接可達物件。只有支援垃圾回收的物件支援 `tp_traverse` 方法，🄵且此方法只會🄵覽涉及參照🄵圈的物件。因此，可以有以下例子：一個整數對於一個引數是直接可達的，這個整數物件有可能出現或不出現在結果的 list 當中。

引發一個附帶引數 `objs` 的稽核事件 `gc.get_referents`。

gc.**is_tracked**(*obj*)

當物件正在被垃圾回收器追🄵時回傳 `True`，否則回傳 `False`。一般來🄵，原子型🄵 (atomic type) 的實例不會被追🄵，而非原子型🄵（如容器、使用者自己定義的物件）會被追🄵。然而，有一些特定型🄵最佳化會被用來🄵少垃圾回收器在簡單實例（如只含有原子性的鍵和值的字典）上的足🄵：

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
True
```

在 3.1 版被加入.

gc.**is_finalized**(*obj*)

如果給定物件已被垃圾回收器終結則回傳 `True`，否則回傳 `False`。:

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

在 3.9 版被加入.

gc.**freeze**()

凍結 (freeze) 垃圾回收器所追🄵的所有物件；將它們移至永久代🄵忽略所有未來的收集動作。

如果一個行程將在🄵有 `exec()` 的情🄵下進行 `fork()`，避免子行程中不必要的寫入時🄵🄵將最大化記憶體共享🄵🄵少整體記憶體使用。這需要避免在父行程的記憶體頁面中建立已釋放的「漏洞」，🄵確保子行程中的 GC 收集不會觸及源自父行程的長壽命物件的 `gc_refs` 計數器。要實現這兩個目標，請在父行程的早期呼叫 `gc.disable()`，在 `fork()` 之前呼叫 `gc.freeze()`，🄵🄵早在子行程中呼叫 `gc.enable()`。

在 3.7 版被加入.

`gc.`**`unfreeze`**`()`

解凍 (unfreeze) 永久代中的物件，🄵將它們放回到最年老代中。

在 3.7 版被加入.

`gc.`**`get_freeze_count`**`()`

回傳永久代中的物件數量。

在 3.7 版被加入.

以下變數僅供唯讀存取（你可以修改其值但不應該重新🄵結 (rebind) 它們）：

`gc.`**`garbage`**

一個回收器發現不可達而又無法被釋放的物件（不可回收物件）list。從 Python 3.4 開始，該 list 在大多數時候都應該是空的，除非使用了有非 `NULL` `tp_del` 槽位的 C 擴充套件型🄵的實例。

如果設定了*`DEBUG_SAVEALL`*，則所有不可達物件將被加進該 list 而不會被釋放。

在 3.2 版的變更: 當*interpreter shutdown* 即直譯器關閉時，若此 list 非空，會🄵生*`ResourceWarning`*，在預設情🄵下此警告不會被提醒。如果設定了*`DEBUG_UNCOLLECTABLE`*，所有無法被回收的物件會被印出。

在 3.4 版的變更: 根據 **PEP 442**，帶有 `__del__()` method 的物件最終不會在*`gc.garbage`* 🄵。

`gc.`**`callbacks`**

會被垃圾回收器在回收開始前和完成後叫用的一系列回呼函式 (callback) 。這些回呼函式在被呼叫時附帶兩個引數：*phase* 和 *info*。

phase 可🄵以下兩者之一：

”start”：垃圾回收即將開始。

”stop”：垃圾回收已結束。

info 是一個字典，提供回呼函式更多資訊。已有定義的鍵有：

”generation”（代）：正在被回收的最年老的一代。

”collected”（已回收的）：當 *phase* 🄵”stop” 時，被成功回收的物件的數目。

”uncollectable”（不可回收的）：當 *phase* 🄵”stop” 時，不能被回收🄵被放入*`garbage`* 的物件的數目。

應用程式可以把他們自己的回呼函式加入此 list。主要的使用場景有：

收集垃圾回收的統計資料，如：不同代的回收頻率、回收任務所花費的時間。

讓應用程式可以識🄵和清理他們自己在*`garbage`* 中的不可回收型🄵物件。

在 3.3 版被加入.

以下常數是🄵了和*`set_debug()`* 一起使用所提供：

`gc.`**`DEBUG_STATS`**

在回收完成後印出統計資訊。當調校回收頻率設定時，這些資訊會很有用。

`gc.`**`DEBUG_COLLECTABLE`**

當發現可回收物件時印出資訊。

gc.**DEBUG_UNCOLLECTABLE**

印出找到的不可回收物件的資訊（指不能被回收器回收的不可達物件）。這些物件會被新增到 garbage list 中。

在 3.2 版的變更: 當*interpreter shutdown*（直譯器關閉）時，若*garbage* list 不是空的，那這些🄵容也會被印出。

gc.**DEBUG_SAVEALL**

設定後，所有回收器找到的不可達物件會被加進 *garbage* 而不是直接被釋放。這在🄵一個記憶體流失的程式除錯時會很有用。

gc.**DEBUG_LEAK**

要印出記憶體流失程式之相關資訊時，回收器所需的除錯旗標。（等同於 DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL）。

29.14 inspect --- 檢視活動物件

原始碼： Lib/inspect.py

The *inspect* module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

29.14.1 Types and members

The *getmembers()* function retrieves the members of an object such as a class or module. The functions whose names begin with ”is” are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes (see import-mod-attrs for module attributes):

Type	屬性	描述
class	__doc__	documentation string
	__name__	name with which this class was defined
	__qualname__	qualified name
	__module__	name of module in which this class was defined
	__type_params__	A tuple containing the type parameters of a generic class
method	__doc__	documentation string
	__name__	name with which this method was defined
	__qualname__	qualified name
	__func__	function object containing implementation of method
	__self__	instance to which this method is bound, or None
	__module__	name of module in which this method was defined
函式	__doc__	documentation string
	__name__	name with which this function was defined
	__qualname__	qualified name
	__code__	code object containing compiled function *bytecode*
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__builtins__	builtins namespace
	__annotations__	mapping of parameters names to annotations; "return" key is reserved for return annota

表格 2 – 繼續上一頁

Type	屬性	描述
	__type_params__	A tuple containing the type parameters of a generic function
	__module__	name of module in which this function was defined
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_generator	returns the generator or coroutine object that owns this frame, or `None` if the frame is of a
	f_trace	tracing function for this frame, or `None`
	f_trace_lines	indicate whether a tracing event is triggered for each source source line
	f_trace_opcodes	indicate whether per-opcode events are requested
	clear()	used to clear all references to local variables
code（程式碼）	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more *here*
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_posonlyargcount	number of positional only arguments
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_qualname	fully qualified name with which this code object was defined
	co_names	tuple of names other than arguments and function locals
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
	co_lines()	returns an iterator that yields successive bytecode ranges
	co_positions()	returns an iterator of source code positions for each bytecode instruction
	replace()	returns a copy of the code object with new values
generator	__name__	name
	__qualname__	qualified name
	gi_frame	frame
	gi_running	is the generator running?
	gi_suspended	is the generator suspended?
	gi_code	code（程式碼）
	gi_yieldfrom	object being iterated by `yield from`, or `None`
async generator	__name__	name
	__qualname__	qualified name
	ag_await	object being awaited on, or `None`
	ag_frame	frame
	ag_running	is the generator running?
	ag_code	code（程式碼）
coroutine	__name__	name
	__qualname__	qualified name
	cr_await	object being awaited on, or `None`
	cr_frame	frame

表格 2 – 繼續上一頁

Type	屬性	描述
	cr_running	is the coroutine running?
	cr_code	code（程式碼）
	cr_origin	where coroutine was created, or `None`. See *sys.set_coroutine_origin_tracking_*
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	qualified name
	__self__	instance to which a method is bound, or `None`

在 3.5 版的變更: 將 `__qualname__` 和 `gi_yieldfrom` 屬性加到🄵生器。

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

在 3.7 版的變更: 新增協程的 `cr_origin` 屬性。

在 3.10 版的變更: 新增函式的 `__builtins__` 屬性。

在 3.14 版的變更: 新增 `f_generator` 屬性到 frame。

`inspect.`**`getmembers`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name. If the optional *predicate* argument—which will be called with the `value` object of each member—is supplied, only members for which the predicate returns a true value are included.

> **備🄵**
>
> *getmembers()* will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmembers_static`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name without triggering dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__. Optionally, only return members that satisfy a given predicate.

> **備🄵**
>
> *getmembers_static()* may not be able to retrieve all members that getmembers can fetch (like dynamically created attributes) and may find members that getmembers can't (like descriptors that raise AttributeError). It can also return descriptor objects instead of instance members in some cases.

在 3.11 版被加入.

`inspect.`**`getmodulename`**(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *importlib.machinery.all_suffixes()*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

在 3.3 版的變更: 此函式直接基於*importlib*。

`inspect.`**`ismodule`**(*object*)

如果物件是模組，則回傳 `True`。

`inspect.`**`isclass`**(*object*)

Return `True` if the object is a class, whether built-in or created in Python code.

inspect.**ismethod**(*object*)

Return `True` if the object is a bound method written in Python.

inspect.**ispackage**(*object*)

如果物件是*package*，則回傳 `True`。

在 3.14 版被加入.

inspect.**isfunction**(*object*)

Return `True` if the object is a Python function, which includes functions created by a *lambda* expression.

inspect.**isgeneratorfunction**(*object*)

如果物件是 Python 🄵生器函式，則回傳 `True`。

在 3.8 版的變更: Functions wrapped in *`functools.partial()`* now return `True` if the wrapped function is a Python generator function.

在 3.13 版的變更: Functions wrapped in *`functools.partialmethod()`* now return `True` if the wrapped function is a Python generator function.

inspect.**isgenerator**(*object*)

如果物件是🄵生器，則回傳 `True`。

inspect.**iscoroutinefunction**(*object*)

Return `True` if the object is a *coroutine function* (a function defined with an `async def` syntax), a *`functools.partial()`* wrapping a *coroutine function*, or a sync function marked with *`markcoroutinefunction()`*.

在 3.5 版被加入.

在 3.8 版的變更: Functions wrapped in *`functools.partial()`* now return `True` if the wrapped function is a *coroutine function*.

在 3.12 版的變更: Sync functions marked with *`markcoroutinefunction()`* now return `True`.

在 3.13 版的變更: Functions wrapped in *`functools.partialmethod()`* now return `True` if the wrapped function is a *coroutine function*.

inspect.**markcoroutinefunction**(*func*)

Decorator to mark a callable as a *coroutine function* if it would not otherwise be detected by *`iscoroutinefunction()`*.

This may be of use for sync functions that return a *coroutine*, if the function is passed to an API that requires *`iscoroutinefunction()`*.

When possible, using an `async def` function is preferred. Also acceptable is calling the function and testing the return with *`iscoroutine()`*.

在 3.12 版被加入.

inspect.**iscoroutine**(*object*)

Return `True` if the object is a *coroutine* created by an `async def` function.

在 3.5 版被加入.

inspect.**isawaitable**(*object*)

Return `True` if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
```

(繼續下一頁)

(繼續上一頁)

```
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

在 3.5 版被加入.

inspect.**isasyncgenfunction**(*object*)

Return `True` if the object is an *asynchronous generator* function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

在 3.6 版被加入.

在 3.8 版的變更: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is an *asynchronous generator* function.

在 3.13 版的變更: Functions wrapped in *functools.partialmethod()* now return `True` if the wrapped function is a *coroutine function*.

inspect.**isasyncgen**(*object*)

Return `True` if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

在 3.6 版被加入.

inspect.**istraceback**(*object*)

Return `True` if the object is a traceback.

inspect.**isframe**(*object*)

Return `True` if the object is a frame.

inspect.**iscode**(*object*)

如果物件是程式碼，則回傳 `True`。

inspect.**isbuiltin**(*object*)

Return `True` if the object is a built-in function or a bound built-in method.

inspect.**ismethodwrapper**(*object*)

Return `True` if the type of object is a *MethodWrapperType*.

These are instances of *MethodWrapperType*, such as `__str__()`, `__eq__()` and `__repr__()`.

在 3.11 版被加入.

inspect.**isroutine**(*object*)

如果物件是使用者定義或🄵建的函式或方法，則回傳 `True`。

inspect.**isabstract**(*object*)

如果物件是抽象基底類🄴，則回傳 `True`。

inspect.**ismethoddescriptor**(*object*)

Return `True` if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method, but not a `__set__()` method or a `__delete__()` method. Beyond that, the set of attributes varies. A *__name__* attribute is usually sensible, and *__doc__* often is.

Methods implemented via descriptors that also pass one of the other tests return `False` from the *ismethoddescriptor()* test, simply because the other tests promise more -- you can, e.g., count on having the `__func__` attribute (etc) when an object passes *ismethod()*.

在 3.13 版的變更: This function no longer incorrectly reports objects with `__get__()` and `__delete__()`, but not `__set__()`, as being method descriptors (such objects are data descriptors, not method descriptors).

`inspect.`**`isdatadescriptor`**(*object*)

如果物件是資料描述器，則回傳 `True`。

Data descriptors have a `__set__` or a `__delete__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *__name__* and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

`inspect.`**`isgetsetdescriptor`**(*object*)

Return `True` if the object is a getset descriptor.

CPython 實作細節： getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

`inspect.`**`ismemberdescriptor`**(*object*)

如果物件是成員描述器，則回傳 `True`。

CPython 實作細節： Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

29.14.2 取得原始碼

`inspect.`**`getdoc`**(*object*)

Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy. Return `None` if the documentation string is invalid or missing.

在 3.5 版的變更: Documentation strings are now inherited if not overridden.

`inspect.`**`getcomments`**(*object*)

Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return `None`. This could happen if the object has been defined in C or the interactive shell.

`inspect.`**`getfile`**(*object*)

Return the name of the (text or binary) file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

`inspect.`**`getmodule`**(*object*)

Try to guess which module an object was defined in. Return `None` if the module cannot be determined.

`inspect.`**`getsourcefile`**(*object*)

Return the name of the Python source file in which an object was defined or `None` if no way can be identified to get the source. This will fail with a *TypeError* if the object is a built-in module, class, or function.

`inspect.`**`getsourcelines`**(*object*)

Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

在 3.3 版的變更: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**getsource**(*object*)

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *OSError* is raised if the source code cannot be retrieved. A *TypeError* is raised if the object is a built-in module, class, or function.

在 3.3 版的變更: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**cleandoc**(*doc*)

Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

29.14.3 Introspecting callables with the Signature object

在 3.3 版被加入.

The *Signature* object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the signature() function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*, *annotation_format=Format.VALUE*)

Return a *Signature* object for the given *callable*:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'

>>> str(sig.parameters['b'])
'b: int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of Python callables, from plain functions and classes to *functools.partial()* objects.

If some of the annotations are strings (e.g., because from __future__ import annotations was used), *signature()* will attempt to automatically un-stringize the annotations using *annotationlib.get_annotations()*. The *globals*, *locals*, and *eval_str* parameters are passed into annotationlib.get_annotations() when resolving the annotations; see the documentation for annotationlib.get_annotations() for instructions on how to use these parameters. A member of the *annotationlib.Format* enum can be passed to the *annotation_format* parameter to control the format of the returned annotations. For example, use annotation_format=annotationlib.Format.STRING to return annotations in string format.

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported. Also, if the annotations are stringized, and *eval_str* is not false, the eval() call(s) to un-stringize the annotations in *annotationlib.get_annotations()* could potentially raise any kind of exception.

A slash (/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

在 3.5 版的變更: The *follow_wrapped* parameter was added. Pass False to get a signature of *callable* specifically (callable.__wrapped__ will not be used to unwrap decorated callables.)

在 3.10 版的變更: The *globals*, *locals*, and *eval_str* parameters were added.

在 3.14 版的變更: The *annotation_format* parameter was added.

> **備🄵**
>
> Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

CPython 實作細節： If the passed object has a `__signature__` attribute, we may use it to create the signature. The exact semantics are an implementation detail and are subject to unannounced changes. Consult the source code for current semantics.

class `inspect.`**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

A `Signature` object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a *`Parameter`* object in its *`parameters`* collection.

The optional *parameters* argument is a sequence of *`Parameter`* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument can be an arbitrary Python object. It represents the "return" annotation of the callable.

`Signature` objects are *immutable*. Use *`Signature.replace()`* or *`copy.replace()`* to make a modified copy.

在 3.5 版的變更: `Signature` objects are now picklable and *hashable*.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters' names to the corresponding *`Parameter`* objects. Parameters appear in strict definition order, including keyword-only parameters.

在 3.7 版的變更: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation

The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to *`Signature.empty`*.

bind(**args*, ***kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns *`BoundArguments`* if `*args` and `**kwargs` match the signature, or raises a *`TypeError`*.

bind_partial(**args*, ***kwargs*)

Works the same way as *`Signature.bind()`*, but allows the omission of some required arguments (mimics *`functools.partial()`* behavior.) Returns *`BoundArguments`*, or raises a *`TypeError`* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)

Create a new *`Signature`* instance based on the instance *`replace()`* was invoked on. It is possible to pass different *parameters* and/or *return_annotation* to override the corresponding properties of the base signature. To remove `return_annotation` from the copied `Signature`, pass in *`Signature.empty`*.

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

Signature objects are also supported by the generic function *copy.replace()*.

format(*, *max_width=None*, *quote_annotation_strings=True*)

Create a string representation of the *Signature* object.

If *max_width* is passed, the method will attempt to fit the signature into lines of at most *max_width* characters. If the signature is longer than *max_width*, all parameters will be on separate lines.

If *quote_annotation_strings* is False, *annotations* in the signature are displayed without opening and closing quotation marks if they are strings. This is useful if the signature was created with the *STRING* format or if `from __future__ import annotations` was used.

在 3.13 版被加入.

在 3.14 版的變更: The *unquote_annotations* parameter was added.

classmethod from_callable(*obj*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

Return a *Signature* (or its subclass) object for a given callable *obj*.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

Its behavior is otherwise identical to that of *signature()*.

在 3.5 版被加入.

在 3.10 版的變更: The *globals*, *locals*, and *eval_str* parameters were added.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` objects are *immutable*. Instead of modifying a `Parameter` object, you can use *Parameter.replace()* or *copy.replace()* to create a modified copy.

在 3.5 版的變更: Parameter objects are now picklable and *hashable*.

empty

A special class-level marker to specify absence of default values and annotations.

name

The name of the parameter as a string. The name must be a valid Python identifier.

CPython 實作細節： CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

在 3.6 版的變更: These parameter names are now exposed by this module as names like `implicit0`.

default

The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation

The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind

Describes how argument values are bound to the parameter. The possible values are accessible via *Parameter* (like `Parameter.KEYWORD_ONLY`), and support comparison and ordering, in the following order:

名懲	意義
POSITIONAL_ONLY	Value must be supplied as a positional argument. Positional only parameters are those which appear before a `/` entry (if present) in a Python function definition.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`kind.`**`description`**

Describes an enum value of *`Parameter.kind`*.

在 3.8 版被加入.

範例：列印所有引數的描述：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

`replace`(**[, name][, kind][, default][, annotation]*)

Create a new *`Parameter`* instance based on the instance replaced was invoked on. To override a `Parameter` attribute, pass the corresponding argument. To remove a default value or/and an annotation from a `Parameter`, pass *`Parameter.empty`*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'
```

(繼續下一頁)

(繼續上一頁)

```
>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

Parameter objects are also supported by the generic function *copy.replace()*.

在 3.4 版的變更: In Python 3.3 *Parameter* objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class inspect.**BoundArguments**

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

A mutable mapping of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

> **備🄵**
>
> Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

在 3.9 版的變更: *arguments* is now of type *dict*. Formerly, it was of type *collections.OrderedDict*.

args

A tuple of positional arguments values. Dynamically computed from the *arguments* attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the *arguments* attribute. Arguments that can be passed positionally are included in *args* instead.

signature

A reference to the parent *Signature* object.

apply_defaults()

🄵遺漏的引數設定預設值。

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

在 3.5 版被加入.

The *args* and *kwargs* properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

也參考

PEP 362 - Function Signature Object.
The detailed specification, implementation details and examples.

29.14.4 類🄵與函式

`inspect.`**`getclasstree`**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.`**`getfullargspec`**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that *`signature()`* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

在 3.4 版的變更: This function is now based on *`signature()`*, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

在 3.6 版的變更: This method was previously documented as deprecated in favour of *`signature()`* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy `getargspec()` API.

在 3.7 版的變更: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

`inspect.`**`getargvalues`**(*frame*)

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

備🄵

This function was inadvertently marked as deprecated in Python 3.5.

`inspect.`**`formatargvalues`**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by *`getargvalues()`*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

> **備🄵**
>
> This function was inadvertently marked as deprecated in Python 3.5.

inspect.**getmro**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.**getcallargs**(*func*, */*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

在 3.2 版被加入.

在 3.5 版之後被🄵用: 請改用*Signature.bind()* 與*Signature.bind_partial()*。

inspect.**getclosurevars**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

如果 *func* 不是 Python 函式或方法，則引發*TypeError*。

在 3.3 版被加入.

inspect.**unwrap**(*func*, *, *stop=None*)

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, *signature()* uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

如果遇到循環，則引發*ValueError*。

在 3.4 版被加入.

inspect.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=annotationlib.Format.VALUE*)

Compute the annotations dict for an object.

This is an alias for *annotationlib.get_annotations()*; see the documentation of that function for more information.

> **警示**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

在 3.10 版被加入.

在 3.14 版的變更: This function is now an alias for *annotationlib.get_annotations()*. Calling it as `inspect.get_annotations` will continue to work.

29.14.5 直譯器堆🄵

Some of the following functions return *FrameInfo* objects. For backwards compatibility these objects allow tuple-like operations on all attributes except `positions`. This behavior is considered deprecated and may be removed in the future.

class inspect.**FrameInfo**

frame
The frame object that the record corresponds to.

filename
The file name associated with the code being executed by the frame this record corresponds to.

lineno
The line number of the current line associated with the code being executed by the frame this record corresponds to.

function
The function name that is being executed by the frame this record corresponds to.

code_context
A list of lines of context from the source code that's being executed by the frame this record corresponds to.

index
The index of the current line being executed in the *code_context* list.

positions
A *dis.Positions* object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this record corresponds to.

在 3.5 版的變更: Return a *named tuple* instead of a *tuple*.

在 3.11 版的變更: `FrameInfo` is now a class instance (that is backwards compatible with the previous *named tuple*).

class inspect.**Traceback**

filename
The file name associated with the code being executed by the frame this traceback corresponds to.

lineno
The line number of the current line associated with the code being executed by the frame this traceback corresponds to.

function
The function name that is being executed by the frame this traceback corresponds to.

code_context

A list of lines of context from the source code that's being executed by the frame this traceback corresponds to.

index

The index of the current line being executed in the *code_context* list.

positions

A *dis.Positions* object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this traceback corresponds to.

在 3.11 版的變更: `Traceback` is now a class instance (that is backwards compatible with the previous *named tuple*).

備🄵

Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using *gc.disable()*. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

Get information about a frame or traceback object. A *Traceback* object is returned.

在 3.11 版的變更: A *Traceback* object is returned instead of a named tuple.

`inspect.`**`getouterframes`**(*frame*, *context=1*)

Get a list of *FrameInfo* objects for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

在 3.5 版的變更: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

在 3.11 版的變更: 回傳一個*FrameInfo* 物件串列。

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

Get a list of *FrameInfo* objects for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

在 3.5 版的變更: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

在 3.11 版的變更: 回傳一個*FrameInfo* 物件串列。

inspect.**currentframe**()

Return the frame object for the caller's stack frame.

CPython 實作細節： This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

inspect.**stack**(*context=1*)

Return a list of *FrameInfo* objects for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

在 3.5 版的變更: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

在 3.11 版的變更: 回傳一個*FrameInfo* 物件串列。

inspect.**trace**(*context=1*)

Return a list of *FrameInfo* objects for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

在 3.5 版的變更: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

在 3.11 版的變更: 回傳一個*FrameInfo* 物件串列。

29.14.6 Fetching attributes statically

Both *getattr()* and *hasattr()* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *getattr_static()* has the same signature as *getattr()* but avoids executing code when it fetches attributes.

inspect.**getattr_static**(*obj*, *attr*, *default=None*)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance `__dict__` is shadowed by another member (for example a property) then this function will be unable to find instance members.

在 3.2 版被加入.

getattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)
```

(繼續下一頁)

(繼續上一頁)

```
result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.14.7 Current State of Generators, Coroutines, and Asynchronous Generators

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

inspect.**getgeneratorstate**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

在 3.2 版被加入.

inspect.**getcoroutinestate**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

在 3.5 版被加入.

inspect.**getasyncgenstate**(*agen*)

Get current state of an asynchronous generator object. The function is intended to be used with asynchronous iterator objects created by `async def` functions which use the `yield` statement, but will accept any asynchronous generator-like object that has `ag_running` and `ag_frame` attributes.

Possible states are:

- AGEN_CREATED: 等待開始執行。
- AGEN_RUNNING: 目前正在被直譯器執行。
- AGEN_SUSPENDED: 目前於 yield 運算式暫停。
- AGEN_CLOSED: 執行已完成。

在 3.12 版被加入.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.**getgeneratorlocals**(*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython 實作細節： This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

在 3.3 版被加入.

inspect.**getcoroutinelocals**(*coroutine*)

This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

在 3.5 版被加入.

inspect.**getasyncgenlocals**(*agen*)

This function is analogous to *getgeneratorlocals()*, but works for asynchronous generator objects created by `async def` functions which use the `yield` statement.

在 3.12 版被加入.

29.14.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.**CO_OPTIMIZED**

The code object is optimized, using fast locals.

inspect.**CO_NEWLOCALS**

If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

inspect.**CO_VARARGS**

The code object has a variable positional parameter (`*args`-like).

inspect.**CO_VARKEYWORDS**

The code object has a variable keyword parameter (`**kwargs`-like).

inspect.**CO_NESTED**

The flag is set when the code object is a nested function.

inspect.**CO_GENERATOR**

The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.**CO_COROUTINE**

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

在 3.5 版被加入.

inspect.**CO_ITERABLE_COROUTINE**

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

在 3.5 版被加入.

inspect.**CO_ASYNC_GENERATOR**

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

在 3.6 版被加入.

inspect.**CO_HAS_DOCSTRING**

The flag is set when there is a docstring for the code object in the source code. If set, it will be the first item in `co_consts`.

在 3.14 版被加入.

inspect.**CO_METHOD**

The flag is set when the code object is a function defined in class scope.

在 3.14 版被加入.

> **備🄵**
>
> The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

29.14.9 Buffer flags

class inspect.**BufferFlags**

This is an *enum.IntFlag* that represents the flags that can be passed to the `__buffer__()` method of objects implementing the buffer protocol.

The meaning of the flags is explained at buffer-request-types.

SIMPLE

WRITABLE

FORMAT

ND

STRIDES

C_CONTIGUOUS

F_CONTIGUOUS

ANY_CONTIGUOUS

INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

在 3.12 版被加入.

29.14.10 命令列介面

The `inspect` module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details

Print information about the specified object rather than the source code

29.15 `annotationlib` --- Functionality for introspecting annotations

在 3.14 版被加入.

Source code: Lib/annotationlib.py

The `annotationlib` module provides tools for introspecting *annotations* on modules, classes, and functions.

Annotations are lazily evaluated and often contain forward references to objects that are not yet defined when the annotation is created. This module provides a set of low-level tools that can be used to retrieve annotations in a reliable way, even in the presence of forward references and other edge cases.

This module supports retrieving annotations in three main formats (see `Format`), each of which works best for different use cases:

- `VALUE` evaluates the annotations and returns their value. This is most straightforward to work with, but it may raise errors, for example if the annotations contain references to undefined names.
- `FORWARDREF` returns `ForwardRef` objects for annotations that cannot be resolved, allowing you to inspect the annotations without evaluating them. This is useful when you need to work with annotations that may contain unresolved forward references.
- `STRING` returns the annotations as a string, similar to how it would appear in the source file. This is useful for documentation generators that want to display annotations in a readable way.

The `get_annotations()` function is the main entry point for retrieving annotations. Given a function, class, or module, it returns an annotations dictionary in the requested format. This module also provides functionality for working directly with the *annotate function* that is used to evaluate annotations, such as `get_annotate_from_class_namespace()` and `call_annotate_function()`, as well as the `call_evaluate_function()` function for working with *evaluate functions*.

警示

Most functionality in this module can execute arbitrary code; see *the security section* for more information.

也參考

PEP 649 proposed the current model for how annotations work in Python.

PEP 749 expanded on various aspects of **PEP 649** and introduced the `annotationlib` module.

annotations-howto provides best practices for working with annotations.

typing-extensions provides a backport of *get_annotations()* that works on earlier versions of Python.

29.15.1 Annotation semantics

The way annotations are evaluated has changed over the history of Python 3, and currently still depends on a future import. There have been execution models for annotations:

- *Stock semantics* (default in Python 3.0 through 3.13; see **PEP 3107** and **PEP 526**): Annotations are evaluated eagerly, as they are encountered in the source code.
- *Stringified annotations* (used with `from __future__ import annotations` in Python 3.7 and newer; see **PEP 563**): Annotations are stored as strings only.
- *Deferred evaluation* (default in Python 3.14 and newer; see **PEP 649** and **PEP 749**): Annotations are evaluated lazily, only when they are accessed.

As an example, consider the following program:

```
def func(a: Cls) -> None:
    print(a)

class Cls: pass

print(func.__annotations__)
```

This will behave as follows:

- Under stock semantics (Python 3.13 and earlier), it will throw a *NameError* at the line where `func` is defined, because `Cls` is an undefined name at that point.
- Under stringified annotations (if `from __future__ import annotations` is used), it will print `{'a': 'Cls', 'return': 'None'}`.
- Under deferred evaluation (Python 3.14 and later), it will print `{'a': <class 'Cls'>, 'return': None}`.

Stock semantics were used when function annotations were first introduced in Python 3.0 (by **PEP 3107**) because this was the simplest, most obvious way to implement annotations. The same execution model was used when variable annotations were introduced in Python 3.6 (by **PEP 526**). However, stock semantics caused problems when using annotations as type hints, such as a need to refer to names that are not yet defined when the annotation is encountered. In addition, there were performance problems with executing annotations at module import time. Therefore, in Python 3.7, **PEP 563** introduced the ability to store annotations as strings using the `from __future__ import annotations` syntax. The plan at the time was to eventually make this behavior the default, but a problem appeared: stringified annotations are more difficult to process for those who introspect annotations at runtime. An alternative proposal, **PEP 649**, introduced the third execution model, deferred evaluation, and was implemented in Python 3.14. Stringified annotations are still used if `from __future__ import annotations` is present, but this behavior will eventually be removed.

29.15.2 Classes

class annotationlib.**Format**

An *IntEnum* describing the formats in which annotations can be returned. Members of the enum, or their equivalent integer values, can be passed to *get_annotations()* and other functions in this module, as well as to `__annotate__` functions.

VALUE = 1

Values are the result of evaluating the annotation expressions.

VALUE_WITH_FAKE_GLOBALS = 2

Special value used to signal that an annotate function is being evaluated in a special environment with fake globals. When passed this value, annotate functions should either return the same value as for the *Format.VALUE* format, or raise *NotImplementedError* to signal that they do not support execution

in this environment. This format is only used internally and should not be passed to the functions in this module.

FORWARDREF = 3

Values are real annotation values (as per *Format.VALUE* format) for defined values, and *ForwardRef* proxies for undefined values. Real objects may contain references to *ForwardRef* proxy objects.

STRING = 4

Values are the text string of the annotation as it appears in the source code, up to modifications including, but not restricted to, whitespace normalizations and constant values optimizations.

The exact values of these strings may change in future versions of Python.

在 3.14 版被加入.

class annotationlib.**ForwardRef**

A proxy object for forward references in annotations.

Instances of this class are returned when the *FORWARDREF* format is used and annotations contain a name that cannot be resolved. This can happen when a forward reference is used in an annotation, such as when a class is referenced before it is defined.

__forward_arg__

A string containing the code that was evaluated to produce the *ForwardRef*. The string may not be exactly equivalent to the original source.

evaluate(*, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=Format.VALUE*)

Evaluate the forward reference, returning its value.

If the *format* argument is *VALUE* (the default), this method may throw an exception, such as *NameError*, if the forward reference refers to a name that cannot be resolved. The arguments to this method can be used to provide bindings for names that would otherwise be undefined. If the *format* argument is *FORWARDREF*, the method will never throw an exception, but may return a *ForwardRef* instance. For example, if the forward reference object contains the code `list[undefined]`, where `undefined` is a name that is not defined, evaluating it with the *FORWARDREF* format will return `list[ForwardRef('undefined')]`. If the *format* argument is *STRING*, the method will return *__forward_arg__*.

The *owner* parameter provides the preferred mechanism for passing scope information to this method. The owner of a *ForwardRef* is the object that contains the annotation from which the *ForwardRef* derives, such as a module object, type object, or function object.

The *globals*, *locals*, and *type_params* parameters provide a more precise mechanism for influencing the names that are available when the *ForwardRef* is evaluated. *globals* and *locals* are passed to *eval()*, representing the global and local namespaces in which the name is evaluated. The *type_params* parameter is relevant for objects created using the native syntax for generic classes and functions. It is a tuple of type parameters that are in scope while the forward reference is being evaluated. For example, if evaluating a *ForwardRef* retrieved from an annotation found in the class namespace of a generic class `C`, *type_params* should be set to `C.__type_params__`.

ForwardRef instances returned by *get_annotations()* retain references to information about the scope they originated from, so calling this method with no further arguments may be sufficient to evaluate such objects. *ForwardRef* instances created by other means may not have any information about their scope, so passing arguments to this method may be necessary to evaluate them successfully.

If no *owner*, *globals*, *locals*, or *type_params* are provided and the *ForwardRef* does not contain information about its origin, empty globals and locals dictionaries are used.

在 3.14 版被加入.

29.15.3 Functions

annotationlib.**annotations_to_string**(*annotations*)

Convert an annotations dict containing runtime values to a dict containing only strings. If the values are not already strings, they are converted using *type_repr()*. This is meant as a helper for user-provided annotate functions that support the *STRING* format but do not have access to the code creating the annotations.

For example, this is used to implement the *STRING* for *typing.TypedDict* classes created through the functional syntax:

```
>>> from typing import TypedDict
>>> Movie = TypedDict("movie", {"name": str, "year": int})
>>> get_annotations(Movie, format=Format.STRING)
{'name': 'str', 'year': 'int'}
```

在 3.14 版被加入.

annotationlib.**call_annotate_function**(*annotate*, *format*, *, *owner=None*)

Call the *annotate function* *annotate* with the given *format*, a member of the *Format* enum, and return the annotations dictionary produced by the function.

This helper function is required because annotate functions generated by the compiler for functions, classes, and modules only support the *VALUE* format when called directly. To support other formats, this function calls the annotate function in a special environment that allows it to produce annotations in the other formats. This is a useful building block when implementing functionality that needs to partially evaluate annotations while a class is being constructed.

owner is the object that owns the annotation function, usually a function, class, or module. If provided, it is used in the *FORWARDREF* format to produce a *ForwardRef* object that carries more information.

也參考

PEP 649 contains an explanation of the implementation technique used by this function.

在 3.14 版被加入.

annotationlib.**call_evaluate_function**(*evaluate*, *format*, *, *owner=None*)

Call the *evaluate function* *evaluate* with the given *format*, a member of the *Format* enum, and return the value produced by the function. This is similar to *call_annotate_function()*, but the latter always returns a dictionary mapping strings to annotations, while this function returns a single value.

This is intended for use with the evaluate functions generated for lazily evaluated elements related to type aliases and type parameters:

- *typing.TypeAliasType.evaluate_value()*, the value of type aliases
- *typing.TypeVar.evaluate_bound()*, the bound of type variables
- *typing.TypeVar.evaluate_constraints()*, the constraints of type variables
- *typing.TypeVar.evaluate_default()*, the default value of type variables
- *typing.ParamSpec.evaluate_default()*, the default value of parameter specifications
- *typing.TypeVarTuple.evaluate_default()*, the default value of type variable tuples

owner is the object that owns the evaluate function, such as the type alias or type variable object.

format can be used to control the format in which the value is returned:

```
>>> type Alias = undefined
>>> call_evaluate_function(Alias.evaluate_value, Format.VALUE)
Traceback (most recent call last):
...
```

(繼續下一頁)

(繼續上一頁)

```
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
>>> call_evaluate_function(Alias.evaluate_value, Format.STRING)
'undefined'
```

在 3.14 版被加入.

annotationlib.**get_annotate_from_class_namespace**(*namespace*)

Retrieve the *annotate function* from a class namespace dictionary *namespace*. Return `None` if the namespace does not contain an annotate function. This is primarily useful before the class has been fully created (e.g., in a metaclass); after the class exists, the annotate function can be retrieved with `cls.__annotate__`. See *below* for an example using this function in a metaclass.

在 3.14 版被加入.

annotationlib.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=Format.VALUE*)

Compute the annotations dict for an object.

obj may be a callable, class, module, or other object with `__annotate__` or `__annotations__` attributes. Passing any other object raises *TypeError*.

The *format* parameter controls the format in which annotations are returned, and must be a member of the *Format* enum or its integer equivalent. The different formats work as follows:

- VALUE: `object.__annotations__` is tried first; if that does not exist, the `object.__annotate__` function is called if it exists.
- FORWARDREF: If `object.__annotations__` exists and can be evaluated successfully, it is used; otherwise, the `object.__annotate__` function is called. If it does not exist either, `object.__annotations__` is tried again and any error from accessing it is re-raised.
- STRING: If `object.__annotate__` exists, it is called first; otherwise, `object.__annotations__` is used and stringified using *annotations_to_string()*.

Returns a dict. `get_annotations()` returns a new dict every time it's called; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you:

- If *eval_str* is true, values of type `str` will be un-stringized using *eval()*. This is intended for use with stringized annotations (`from __future__ import annotations`). It is an error to set *eval_str* to true with formats other than *Format.VALUE*.
- If *obj* doesn't have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict; classes, modules, and other types of callables may not.)
- Ignores inherited annotations on classes, as well as annotations on metaclasses. If a class doesn't have its own annotations dict, returns an empty dict.
- All accesses to object members and dict values are done using `getattr()` and `dict.get()` for safety.

eval_str controls whether or not values of type `str` are replaced with the result of calling *eval()* on those values:

- If eval_str is true, *eval()* is called on values of type `str`. (Note that `get_annotations()` doesn't catch exceptions; if *eval()* raises an exception, it will unwind the stack past the `get_annotations()` call.)
- If *eval_str* is false (the default), values of type `str` are unchanged.

globals and *locals* are passed in to *eval()*; see the documentation for *eval()* for more information. If *globals* or *locals* is `None`, this function may replace that value with a context-specific default, contingent on `type(obj)`:

- If *obj* is a module, *globals* defaults to `obj.__dict__`.
- If *obj* is a class, *globals* defaults to `sys.modules[obj.__module__].__dict__` and *locals* defaults to the *obj* class namespace.
- If *obj* is a callable, *globals* defaults to `obj.__globals__`, although if *obj* is a wrapped function (using *functools.update_wrapper()*) or a *functools.partial* object, it is unwrapped until a non-wrapped function is found.

Calling `get_annotations()` is best practice for accessing the annotations dict of any object. See annotations-howto for more information on annotations best practices.

```
>>> def f(a: int, b: str) -> float:
...     pass
>>> get_annotations(f)
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

在 3.14 版被加入.

`annotationlib.`**`type_repr`**(*value*)

Convert an arbitrary Python value to a format suitable for use by the *STRING* format. This calls *repr()* for most objects, but has special handling for some objects, such as type objects.

This is meant as a helper for user-provided annotate functions that support the *STRING* format but do not have access to the code creating the annotations. It can also be used to provide a user-friendly string representation for other objects that contain values that are commonly encountered in annotations.

在 3.14 版被加入.

29.15.4 Recipes

Using annotations in a metaclass

A metaclass may want to inspect or even modify the annotations in a class body during class creation. Doing so requires retrieving annotations from the class namespace dictionary. For classes created with `from __future__ import annotations`, the annotations will be in the `__annotations__` key of the dictionary. For other classes with annotations, *get_annotate_from_class_namespace()* can be used to get the annotate function, and *call_annotate_function()* can be used to call it and retrieve the annotations. Using the *FORWARDREF* format will usually be best, because this allows the annotations to refer to names that cannot yet be resolved when the class is created.

To modify the annotations, it is best to create a wrapper annotate function that calls the original annotate function, makes any necessary adjustments, and returns the result.

Below is an example of a metaclass that filters out all *typing.ClassVar* annotations from the class and puts them in a separate attribute:

```
import annotationlib
import typing

class ClassVarSeparator(type):
   def __new__(mcls, name, bases, ns):
      if "__annotations__" in ns:  # from __future__ import annotations
         annotations = ns["__annotations__"]
         classvar_keys = {
            key for key, value in annotations.items()
            # Use string comparison for simplicity; a more robust solution
            # could use annotationlib.ForwardRef.evaluate
            if value.startswith("ClassVar")
         }
         classvars = {key: annotations[key] for key in classvar_keys}
         ns["__annotations__"] = {
            key: value for key, value in annotations.items()
            if key not in classvar_keys
```

(繼續下一頁)

(繼續上一頁)

```
        }
        wrapped_annotate = None
    elif annotate := annotationlib.get_annotate_from_class_namespace(ns):
        annotations = annotationlib.call_annotate_function(
            annotate, format=annotationlib.Format.FORWARDREF
        )
        classvar_keys = {
            key for key, value in annotations.items()
            if typing.get_origin(value) is typing.ClassVar
        }
        classvars = {key: annotations[key] for key in classvar_keys}

        def wrapped_annotate(format):
            annos = annotationlib.call_annotate_function(annotate, format, owner=typ)
            return {key: value for key, value in annos.items() if key not in classvar_keys}

    else:  # no annotations
        classvars = {}
        wrapped_annotate = None
    typ = super().__new__(mcls, name, bases, ns)

    if wrapped_annotate is not None:
        # Wrap the original __annotate__ with a wrapper that removes ClassVars
        typ.__annotate__ = wrapped_annotate
    typ.classvars = classvars  # Store the ClassVars in a separate attribute
    return typ
```

29.15.5 Limitations of the `STRING` format

The *`STRING`* format is meant to approximate the source code of the annotation, but the implementation strategy used means that it is not always possible to recover the exact source code.

First, the stringifier of course cannot recover any information that is not present in the compiled code, including comments, whitespace, parenthesization, and operations that get simplified by the compiler.

Second, the stringifier can intercept almost all operations that involve names looked up in some scope, but it cannot intercept operations that operate fully on constants. As a corollary, this also means it is not safe to request the `STRING` format on untrusted code: Python is powerful enough that it is possible to achieve arbitrary code execution even with no access to any globals or builtins. For example:

```
>>> def f(x: (1).__class__.__base__.__subclasses__()[-1].__init__.__builtins__["print"](
↪"Hello world")): pass
...
>>> annotationlib.get_annotations(f, format=annotationlib.Format.STRING)
Hello world
{'x': 'None'}
```

> **備🄵**
>
> This particular example works as of the time of writing, but it relies on implementation details and is not guaranteed to work in the future.

Among the different kinds of expressions that exist in Python, as represented by the *`ast`* module, some expressions are supported, meaning that the `STRING` format can generally recover the original source code; others are unsupported, meaning that they may result in incorrect output or an error.

The following are supported (sometimes with caveats):

- *`ast.BinOp`*
- *`ast.UnaryOp`*

  - *ast.Invert* (~), *ast.UAdd* (+), and *ast.USub* (-) are supported
  - *ast.Not* (not) is not supported
- *ast.Dict* (except when using ** unpacking)
- *ast.Set*
- *ast.Compare*
  - *ast.Eq* and *ast.NotEq* are supported
  - *ast.Lt*, *ast.LtE*, *ast.Gt*, and *ast.GtE* are supported, but the operand may be flipped
  - *ast.Is*, *ast.IsNot*, *ast.In*, and *ast.NotIn* are not supported
- *ast.Call* (except when using ** unpacking)
- *ast.Constant* (though not the exact representation of the constant; for example, escape sequences in strings are lost; hexadecimal numbers are converted to decimal)
- *ast.Attribute* (assuming the value is not a constant)
- *ast.Subscript* (assuming the value is not a constant)
- *ast.Starred* (* unpacking)
- *ast.Name*
- *ast.List*
- *ast.Tuple*
- *ast.Slice*

The following are unsupported, but throw an informative error when encountered by the stringifier:

- *ast.FormattedValue* (f-strings; error is not detected if conversion specifiers like !r are used)
- *ast.JoinedStr* (f-strings)

The following are unsupported and result in incorrect output:

- *ast.BoolOp* (and and or)
- *ast.IfExp*
- *ast.Lambda*
- *ast.ListComp*
- *ast.SetComp*
- *ast.DictComp*
- *ast.GeneratorExp*

The following are disallowed in annotation scopes and therefore not relevant:

- *ast.NamedExpr* (:=)
- *ast.Await*
- *ast.Yield*
- *ast.YieldFrom*

29.15.6 Limitations of the FORWARDREF format

The *FORWARDREF* format aims to produce real values as much as possible, with anything that cannot be resolved replaced with *ForwardRef* objects. It is affected by broadly the same Limitations as the *STRING* format: annotations that perform operations on literals or that use unsupported expression types may raise exceptions when evaluated using the *FORWARDREF* format.

Below are a few examples of the behavior with unsupported expressions:

```
>>> from annotationlib import get_annotations, Format
>>> def zerodiv(x: 1 / 0): ...
>>> get_annotations(zerodiv, format=Format.STRING)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> get_annotations(zerodiv, format=Format.FORWARDREF)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def ifexp(x: 1 if y else 0): ...
>>> get_annotations(ifexp, format=Format.STRING)
{'x': '1'}
```

29.15.7 Security implications of introspecting annotations

Much of the functionality in this module involves executing code related to annotations, which can then do arbitrary things. For example, *get_annotations()* may call an arbitrary *annotate function*, and *ForwardRef.evaluate()* may call *eval()* on an arbitrary string. Code contained in an annotation might make arbitrary system calls, enter an infinite loop, or perform any other operation. This is also true for any access of the __annotations__ attribute, and for various functions in the *typing* module that work with annotations, such as *typing.get_type_hints()*.

Any security issue arising from this also applies immediately after importing code that may contain untrusted annotations: importing code can always cause arbitrary operations to be performed. However, it is unsafe to accept strings or other input from an untrusted source and pass them to any of the APIs for introspecting annotations, for example by editing an __annotations__ dictionary or directly creating a *ForwardRef* object.

29.16 site --- Site-specific configuration hook

原始碼： Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's -S option.

Importing this module normally appends site-specific paths to the module search path and adds *callables*, including *help()* to the built-in namespace. However, Python startup option -S blocks this and this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the *main()* function.

在 3.3 版的變更: Importing the module used to trigger paths manipulation even when using -S.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/pythonX.Y[t]/site-packages (on Unix and macOS). (The optional suffix "t" indicates the *free threading* build, and is appended if "t" is present in the *sys.abiflags* constant.) For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

在 3.5 版的變更: Support for the "site-python" directory has been removed.

在 3.13 版的變更: On Unix, *Free threading* Python installations are identified by the "t" suffix in the version-specific directory name, such as lib/python3.13t/.

在 3.14 版的變更: *site* is no longer responsible for updating *sys.prefix* and *sys.exec_prefix* on 🄵擬環境. This is now done during the *path initialization*. As a result, under 🄵擬環境, *sys.prefix* and *sys.exec_prefix* no longer depend on the *site* initialization, and are therefore unaffected by -S. When running under a *virtual environment*, the pyvenv.cfg file in *sys.prefix* is checked for site-specific configurations. If the include-system-site-packages key exists and is set to true (case-insensitive), the system-level prefixes will be searched for site-packages, otherwise they won't.

A path configuration file is a file whose name has the form *name*`.pth` and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` (followed by space or tab) are executed.

> **備🄵**
>
> An executable line in a `.pth` file is run at every Python startup, regardless of whether a particular module is actually going to be used. Its impact should thus be kept to a minimum. The primary intended purpose of executable lines is to make the corresponding module(s) importable (load 3rd-party import hooks, adjust `PATH` etc). Any other initialization is supposed to be done upon a module's actual import, if and when it happens. Limiting a code chunk to a single line is a deliberate measure to discourage putting anything more complex here.

在 3.13 版的變更: The `.pth` files are now decoded by UTF-8 at first and then by the *locale encoding* if it fails.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following:

```
# foo package configuration

foo
bar
bletch
```

and `bar.pth` contains:

```
# bar package configuration

bar
```

Then the following version-specific directories are added to `sys.path`, in this order:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

29.16.1 `sitecustomize`

After these path manipulations, an attempt is made to import a module named *sitecustomize*, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *ImportError* or its subclass exception, and the exception's *name* attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from *sitecustomize* is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

29.16.2 `usercustomize`

After this, an attempt is made to import a module named *usercustomize*, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. If this import fails with an *ImportError* or its subclass exception, and the exception's *name* attribute equals to `'usercustomize'`, it is silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of *sitecustomize* and *usercustomize* is still attempted.

29.16.3 Readline configuration

On systems that support *readline*, this module will also import and configure the *rlcompleter* module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the *sys.__interactivehook__* attribute in your *sitecustomize* or *usercustomize* module or your `PYTHONSTARTUP` file.

在 3.4 版的變更: Activation of rlcompleter and history was made automatic.

29.16.4 模組🄵容

`site.`**`PREFIXES`**

A list of prefixes for site-packages directories.

`site.`**`ENABLE_USER_SITE`**

Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

`site.`**`USER_SITE`**

Path to the user site-packages for the running Python. Can be `None` if *getusersitepackages()* hasn't been called yet. Default value is `~/.local/lib/python`*X.Y*`[t]/site-packages` for UNIX and non-framework macOS builds, `~/Library/Python/`*X.Y*`/lib/python/site-packages` for macOS framework builds, and `%`*APPDATA*`%\Python\Python`*XY*`\site-packages` on Windows. The optional "t" indicates the free-threaded build. This directory is a site directory, which means that `.pth` files in it will be processed.

`site.`**`USER_BASE`**

Path to the base directory for the user site-packages. Can be `None` if *getuserbase()* hasn't been called yet. Default value is `~/.local` for UNIX and macOS non-framework builds, `~/Library/Python/`*X.Y* for macOS framework builds, and `%`*APPDATA*`%\Python` for Windows. This value is used to compute the installation directories for scripts, data files, Python modules, etc. for the *user installation scheme*. See also `PYTHONUSERBASE`.

`site.`**`main`**`()`

Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the `-S` flag.

在 3.3 版的變更: This function used to be called unconditionally.

`site.`**`addsitedir`**`(`*sitedir*, *known_paths=None*`)`

Add a directory to sys.path and process its `.pth` files. Typically used in *sitecustomize* or *usercustomize* (see above).

`site.`**`getsitepackages`**`()`

Return a list containing all global site-packages directories.

在 3.2 版被加入.

`site.`**`getuserbase`**`()`

Return the path of the user base directory, *USER_BASE*. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

在 3.2 版被加入.

`site.`**`getusersitepackages`**`()`

Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting *USER_BASE*. To determine if the user-specific site-packages was added to `sys.path` *ENABLE_USER_SITE* should be used.

在 3.2 版被加入.

29.16.5 命令列介面

The *site* module also provides a way to get the user directories from the command line:

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

If it is called without arguments, it will print the contents of *sys.path* on the standard output, followed by the value of *USER_BASE* and whether the directory exists, then the same thing for *USER_SITE*, and finally the value of *ENABLE_USER_SITE*.

--user-base

Print the path to the user base directory.

--user-site

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by *os.pathsep*.

If any option is given, the script will exit with one of these values: `0` if the user site-packages directory is enabled, `1` if it was disabled by the user, `2` if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

也參考

- **PEP 370** -- Per user site-packages directory
- *sys.path 模組搜尋路徑的初始化* -- The initialization of *sys.path*.

也參考

- 請參🄵*concurrent.interpreters* 模組，它同樣揭露了核心的 runtime 功能。

CHAPTER 30

自訂 Python 直譯器

本章介紹的模組可用於編寫類似於 Python 交互式直譯器的介面。如果你需要一個在 Python 語言基礎上支援某些特殊功能的直譯器，可以參考 *code* 模組。(*codeop* 模組屬於底層工具，用於支援編譯可能不完整的 Python 程式碼。)

本章節所描述的模組清單🄵:

30.1 code --- 直譯器基底類🄵

原始碼： Lib/code.py

`code` 模組提供在 Python 中實作讀取-求值-印出🄵圈（read-eval-print loops）的設施。包含兩個類🄵和便利函式，可用於建立提供互動式直譯器提示的應用程式。

class `code.`**InteractiveInterpreter**(*locals=None*)

這個類🄵處理剖析和直譯器狀態（使用者的命名空間）；它不處理輸入緩衝、提示或輸入檔案命名（檔案名稱總是明確傳入）。可選的 *locals* 引數指定一個對映，作🄵執行程式碼的命名空間；它預設🄵新建立的字典，鍵 `'__name__'` 設🄵 `'__console__'` 而鍵 `'__doc__'` 會設🄵 `None`。

請注意，在 `InteractiveInterpreter` 實例下建立的函式和類🄵物件將屬於由 *locals* 指定的命名空間。只有當 *locals* 是現有模組的命名空間時，它們才會是可被 pickle 的。

class `code.`**InteractiveConsole**(*locals=None*, *filename='<console>'*, *local_exit=False*)

近似地模擬出互動式 Python 直譯器的行🄵。這個類🄵建立在 *InteractiveInterpreter* 的基礎上，🄵加入使用熟悉的 `sys.ps1` 和 `sys.ps2` 的提示，以及輸入緩衝。如果 *local_exit* 🄵 true，控制台中的 `exit()` 和 `quit()` 將不會引發 *SystemExit*，而是回傳到呼叫程式碼。

在 3.13 版的變更: 新增 *local_exit* 參數。

`code.`**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*, *local_exit=False*)

執行 read-eval-print 🄵圈的便利函式。這會建立一個 *InteractiveConsole* 的新實例，🄵設定 *readfunc* 以用於 *InteractiveConsole.raw_input()* 方法（如有提供）。如果 *local* 有被提供，它會被傳給 *InteractiveConsole* 的建構函式以作🄵直譯器🄵圈的預設命名空間。如果有提供 *local_exit*，它會被傳給 *InteractiveConsole* 構建函式。然後實例的 *interact()* 方法會執行，🄵傳入 *banner* 和 *exitmsg* 以作🄵要使用的🄵幅和退出訊息（如有提供）。控制台物件在使用後就會被🄵🄵。

在 3.6 版的變更: 新增 *exitmsg* 參數。

在 3.13 版的變更: 新增 *local_exit* 參數。

code.**compile_command**(*source*, *filename='<input>'*, *symbol='single'*)

這個函式對於想要模擬 Python 的直譯器主🄵圈（即讀取-求值-印出🄵圈）的程式很有用。最棘手的部分是判斷使用者何時輸入了一個不完整的命令，而這個命令可以透過輸入更多文字來完成（相對於完整的命令或語法錯誤）。這個函式幾乎總是做出與真正的直譯器主🄵圈相同的判斷。

source 是來源字串；*filename* 是讀取來源的可選檔案名稱，預設🄵 `'<input>'`；*symbol* 是可選的文法 (grammar) 起始符號，其應🄵 `'single'`（預設值）、`'eval'` 或 `'exec'`。

如果命令完整且有效，則回傳程式碼物件（與 `compile(source,filename,symbol)` 相同）；如果命令不完整，則回傳 `None`；如果命令完整但包含語法錯誤，則引發*SyntaxError*，如果命令包含無效的字面值 (literal)，則引發*OverflowError* 或*ValueError*。

30.1.1 互動式直譯器物件

InteractiveInterpreter.**runsource**(*source*, *filename='<input>'*, *symbol='single'*)

在直譯器中編譯🄵執行某些原始碼。引數與*compile_command()* 相同；*filename* 的預設值是 `'<input>'`，*symbol* 的預設值是 `'single'`。有幾種情🄵會發生：

- 輸入不正確；*compile_command()* 會引發例外（*SyntaxError* 或*OverflowError*）。語法回溯 (syntax traceback) 會透過呼叫*showsyntaxerror()* 方法來印出。*runsource()* 會回傳 `False`。
- 輸入不完整，需要更多輸入；*compile_command()* 回傳了 `None`。*runsource()* 會回傳 `True`。
- 輸入完整；*compile_command()* 回傳一個程式碼物件。程式碼會透過呼叫*runcode()* 執行（它也會處理執行時的例外，除*SystemExit* 以外）。*runsource()* 會回傳 `False`。

回傳值可用來🄵定是使用 `sys.ps1` 還是 `sys.ps2` 以提示下一列。

InteractiveInterpreter.**runcode**(*code*)

執行程式碼物件。當例外發生時，會呼叫*showtraceback()* 來顯示回溯。除了*SystemExit* 允許繼續傳播之外，所有的例外都會被捕獲。

關於*KeyboardInterrupt* 的注意事項：此例外可能發生在此程式碼的其他地方，而且不一定會被捕獲到。呼叫者應該准備好處理它。

InteractiveInterpreter.**showsyntaxerror**(*filename=None*)

顯示剛剛發生的語法錯誤。這不會顯示堆🄵追🄵 (stack trace)，因🄵語法錯誤🄵有堆🄵追🄵。如果給出 *filename*，它會被塞入例外，而不是 Python 剖析器提供的預設檔案名稱，因🄵它從字串讀取時總是使用 `'<string>'`。輸出會由*write()* 方法寫入。

InteractiveInterpreter.**showtraceback**()

顯示剛剛發生的例外。我們移除第一個堆🄵項目，因🄵它在直譯器物件的實作範圍🄵。輸出由*write()* 方法寫入。

在 3.5 版的變更: 會顯示完整的連鎖回溯記🄵，而不只是主要回溯。

InteractiveInterpreter.**write**(*data*)

寫入字串到標準錯誤串流 (`sys.stderr`)。衍生類🄵應覆寫此功能，以根據需求提供適當的輸出處理。

30.1.2 互動式控制台物件

InteractiveConsole 類🄵是*InteractiveInterpreter* 的子類🄵，因此提供了所有直譯器物件的方法以及下列新增的功能。

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)

近似地模擬互動式 Python 控制台。可選的 *banner* 引數指定在第一次互動之前要印出的橫幅；預設會印出類似標準 Python 直譯器所列印的橫幅，接著是控制台物件在括弧中的類🄵名稱 (以免與真正的直譯器混淆 -- 因🄵它是如此接近！)。

可選的 *exitmsg* 引數指定退出時列印的退出訊息。傳遞空字串以抑制退出訊息。如果🄵有給 *exitmsg* 或 `None`，則會印出預設訊息。

在 3.4 版的變更: 若要抑制印出任何🄵幅，請傳入空字串。

在 3.6 版的變更: 退出時印出退出訊息。

`InteractiveConsole.`**`push`**(*line*)

推送一行原始文字到直譯器。這一行不應該有尾部🄵行符號；它可能有🄵部🄵行符號。這行文字會被附加到緩衝區，🄵且直譯器的*runsource()* 方法會被呼叫，🄵以緩衝區的串接🄵容做🄵原始碼。如果這表示命令已執行或無效，緩衝區會被重設；否則，命令會不完整，且緩衝區會保持該行被附加後的樣子。如果需要更多的輸入，回傳值是 `True`，如果該行已經以某種方式處理，回傳值是 `False`（這與 `runsource()` 相同）。

`InteractiveConsole.`**`resetbuffer`**()

從輸入緩衝區移除任何未處理的原始文字。

`InteractiveConsole.`**`raw_input`**(*prompt=''*)

寫入一個提示🄵讀取一行。回傳的行不包括尾部的🄵行符號。當使用者輸入 EOF 鍵序時，會引發*EOFError*。基底實作會從 `sys.stdin` 讀取；子類🄵可以用不同的實作替代。

30.2 `codeop` --- 編譯 Python 程式碼

原始碼： Lib/codeop.py

---

codeop 模組提供了可以模擬 Python read-eval-print 循環的工具程式 (utilities)，就像在*code* 模組中所做的那樣。因此你可能不想直接使用該模組；如果你想在程式中包含這樣的循環，你可能需要使用*code* 模組。

這個任務有兩個部分：

1. 能🄵判斷一列輸入是否完成了一項 Python 陳述式：簡而言之，判斷接下來是列印'>>>' 還是'...'。
2. 記住使用者輸入了哪些未來陳述式，以便後續輸入可以在這些陳述式生效的情🄵下進行編譯。

codeop 模組提供了一種完成上述每項任務的方法，以及同時完成這兩項任務的方法。

只做前者：

`codeop.`**`compile_command`**(*source*, *filename='<input>'*, *symbol='single'*)

嘗試編譯 *source*，它應該是 Python 程式碼的字串，如果 *source* 是有效的 Python 程式碼，則回傳一個程式碼物件 (code object)。在這種情🄵下，程式碼物件的檔案名稱屬性將🄵 *filename*，預設🄵 `'<input>'`。如果 *source* 不是有效的 Python 程式碼，而是有效 Python 程式碼的前綴，則回傳 `None`。

如果 *source* 有問題，就會引發例外。如果存在無效的 Python 語法則會引發*SyntaxError*；如果存在無效的文字 (literal)，則會引發*OverflowError* 或*ValueError*。

symbol 引數🄵定 *source* 是否編譯🄵陳述式（`'single'`，🄵預設值）、🄵陳述式序列 (`'exec'`) 或🄵運算式 (`'eval'`)。任何其他值都會導致引發*ValueError*。

> **備🄵**
>
> 剖析器 (parser) 有可能（但通常不會）在到達原始碼的結尾之前停止剖析🄵獲得成功的結果；在這種情🄵下，尾隨符號可能會被忽略而不是導致錯誤。例如，反斜🄵後面加上兩個🄵行符號後可以是任意的無意義符號。這個問題在未來會因🄵剖析器 API 的改善而被解🄵。

`class` `codeop.`**`Compile`**

此類🄵的實例具有 `__call__()` 方法，其簽名與🄵建函式*compile()* 相同，區🄵在於如果實例編譯包含*__future__* 陳述式的程式文本，實例會「記住」🄵使用該陳述式開始編譯所有後續程式文本。

`class` `codeop.`**`CommandCompiler`**

此類🄵的實例具有 `__call__()` 方法，其簽名與🄵建函式*compile_command()* 相同，區🄵在於如果實例編譯包含*__future__* 陳述式的程式文本，實例會「記住」🄵使用該陳述式開始編譯所有後續程式文本。

CHAPTER 31

引入模組

本章節所描述的模組提供了引入其他 Python 模組的新方法，以及用於自訂引入過程的 hook。

本章節所描述的完整模組列表🄵：

31.1 `zipimport` --- 從 Zip 封存檔案滙入模組

原始碼： Lib/zipimport.py

此模組新增了從 ZIP 格式封存檔案中引入 Python 模組（`*.py`、`*.pyc`）和套件的能力。通常不需要明確地使用 *`zipimport`* 模組；對於 *`sys.path`* 中指向 ZIP 封存檔案路徑的項目，🄵建的 `import` 機制會自動使用它。

通常 *`sys.path`* 是一個由目🄵名稱字串組成的串列。此模組也允許 *`sys.path`* 的一個項目是命名 ZIP 封存檔案的字串。ZIP 封存檔案可以包含子目🄵結構以支援套件引入，🄵且可以指定封存檔案🄵的路徑以僅從子目🄵中引入。例如路徑 `example.zip/lib/` 將只會從封存檔案🄵的 `lib/` 子目🄵中引入。

ZIP 封存檔案中可以存在任何檔案，但引入器只會對 `.py` 和 `.pyc` 檔案起作用。不允許從 ZIP 引入動態模組（`.pyd`、`.so`）。請注意，如果封存檔案只包含 `.py` 檔案，Python 將不會試圖透過新增對應的 `.pyc` 檔案來修改封存檔案，這意味著如果 ZIP 封存檔案不包含 `.pyc` 檔案，引入可能會相當慢。

在 3.13 版的變更: 支援 ZIP64。

在 3.8 版的變更: 在過去，不支援帶有封存檔案🄵解的 ZIP 封存檔案。

也參考

PKZIP Application Note
由 ZIP 格式和所用演算法的創造者 Phil Katz 編寫的關於 ZIP 檔案格式的文件。

PEP 273 - 從 Zip 封存檔案引入模組
由 James C. Ahlstrom 編寫，他也提供了一個實作。Python 2.3 遵循 **PEP 273** 中的規範，但使用由 Just van Rossum 編寫的實作，該實作使用了 **PEP 302** 中描述的引入🄵🄵。

***`importlib`* - 引入機制的實作**
提供所有引入器要實作的相關協定的套件。

此模組定義了一個例外：

exception zipimport.**ZipImportError**

由 zipimporter 物件引發的例外。它是*ImportError* 的子類🄵，所以也可以被捕捉🄵*ImportError*。

31.1.1 zipimporter 物件

zipimporter 是用於引入 ZIP 檔案的類🄵。

class zipimport.**zipimporter**(*archivepath*)

建立一個新的 zipimporter 實例。*archivepath* 必須是 ZIP 檔案的路徑，或是 ZIP 檔案🄵的特定路徑。例如，`foo/bar.zip/lib` 的 *archivepath* 將會在 ZIP 檔案 `foo/bar.zip` 🄵的 `lib` 目🄵中尋找模組（如果該目🄵存在的話）。

如果 *archivepath* 未指向一個有效的 ZIP 封存檔案，則會引發*ZipImportError*。

在 3.12 版的變更: 在 3.10 版中已🄵用的 `find_loader()` 和 `find_module()` 方法現已被移除。請改用*find_spec()*。

create_module(*spec*)

importlib.abc.Loader.create_module() 的實作，回傳*None* 以明確請求預設語意。

在 3.10 版被加入.

exec_module(*module*)

importlib.abc.Loader.exec_module() 的實作。

在 3.10 版被加入.

find_spec(*fullname*, *target=None*)

importlib.abc.PathEntryFinder.find_spec() 的一個實作。

在 3.10 版被加入.

get_code(*fullname*)

回傳指定模組的程式碼物件。如果模組無法被引入，則引發*ZipImportError*。

get_data(*pathname*)

回傳與 *pathname* 相關的資料。如果找不到檔案，則引發*OSError*。

在 3.3 版的變更: 過去會引發*IOError*，現在它是*OSError* 的🄵名。

get_filename(*fullname*)

回傳如果指定模組被引入時 `__file__` 會被設定的值。如果模組無法被引入，則引發*ZipImportError*。

在 3.1 版被加入.

get_source(*fullname*)

回傳指定模組的原始碼。如果找不到模組，則引發*ZipImportError*；如果封存檔案包含該模組但🄵有其原始碼，則回傳*None*。

is_package(*fullname*)

如果由 *fullname* 指定的模組是一個套件，則回傳 `True`。如果找不到模組，則引發*ZipImportError*。

load_module(*fullname*)

載入由 *fullname* 指定的模組。*fullname* 必須是完整限定（以點分隔）的模組名稱。成功時回傳引入的模組，失敗時引發*ZipImportError*。

Deprecated since version 3.10, will be removed in version 3.15: 請改用*exec_module()*。

invalidate_caches()

清除關於在 ZIP 封存檔案中找到的檔案資訊的🄵部快取。

在 3.10 版被加入.

archive

引入器關聯的 ZIP 檔案的檔名，不包含可能的子路徑。

prefix

在 ZIP 檔案🄵搜尋模組的子路徑。對於指向 ZIP 檔案根目🄵的 zipimporter 物件，這是一個空字串。

archive 和*prefix* 屬性（attribute），當用斜🄵組合時，等於給予*zipimporter* 建構函式的原始 *archivepath* 引數。

31.1.2 范例

這是一個從 ZIP 封存檔案引入模組的範例——請注意，*zipimport* 模組🄵未被明確使用。

```
$ unzip -l example_archive.zip
Archive:  example_archive.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  01-01-00 12:30   example.py
 --------                   -------
     8467                   1 file
```

```
>>> import sys
>>> # 將封存檔案新增到系統路徑的最前面
>>> sys.path.insert(0, 'example_archive.zip')
>>> import example
>>> example.__file__
'example_archive.zip/example.py'
```

31.2 pkgutil --- 套件擴充工具程式

原始碼： Lib/pkgutil.py

此模組提供了引入系統 (import system) 的工具程式，特🄵是套件相關支援。

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)

一個包含模組資訊之簡短摘要的附名元組 (namedtuple)。

在 3.6 版被加入.

pkgutil.**extend_path**(*path*, *name*)

擴充組成一個套件之模組的搜尋路徑。預期用法是將以下程式碼放入套件的 `__init__.py`：

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

對於*sys.path* 上具有與套件名稱相符的子目🄵的每個目🄵，將該子目🄵新增至套件的 `__path__` 中。如果想要將單一邏輯套件的不同部分給分配到多個目🄵時，這會非常有用。

它還會尋找 `*.pkg` 檔案，其中開頭的 `*` 與 *name* 引數相符。此功能類似於 `*.pth` 檔案（更多資訊請參🄵*site* 模組），但他不特🄵處理以 `import` 🄵開頭的行。`*.pkg` 檔案從表面上看是受信任的：除了跳過空行和備🄵之外，在 `*.pkg` 檔案中找到的所有條目都將新增到路徑中，無論它們是否存在於檔案系統。（這是一個功能。）

如果輸入路徑不是串列（像是凍結套件 (frozen package) 的情🄵），它將原封不動地被回傳。輸入路徑不會被修改；而是回傳擴充後的副本。僅將項目附加到副本的尾端。

sys.path 被假設是一個序列，*sys.path* 中的項目🄵，若不是代表現存目🄵的字串則將被忽略。*sys.path* 上用作檔案名稱時導致錯誤的 Unicode 項目可能會導致此函式引發例外（與*os.path.isdir()* 行🄵一致）。

pkgutil.**get_importer**(*path_item*)

取得給定之 *path_item* 的*finder*。

如果回傳的尋檢器 (finder) 是由路徑勾點 (path hook) 所新建立的，則它會被快取在*sys.path_importer_cache* 中。

如果需要重新掃描*sys.path_hooks*，可以手動清除快取（或部分快取）。

在 3.3 版的變更: 更新🄵直接基於*importlib*，而不是依賴套件🄵部 **PEP 302** 的引入模擬 (import emulation)。

pkgutil.**iter_importers**(*fullname=''*)

yield 給定模組名稱的*finder* 物件。

如果 *fullname* 包含 `'.'`，則尋檢器將針對包含 *fullname* 的套件，否則它們全部會是在頂層被🄵🄵的尋檢器（即*sys.meta_path* 和*sys.path_hooks* 上的尋檢器）。

如果指定的模組位於套件中，則作🄵呼叫此函式的副作用 (side effect)，該套件會被引入。

如果未指定模組名稱，則會🄵生所有頂層尋檢器。

在 3.3 版的變更: 更新🄵直接基於*importlib*，而不是依賴套件🄵部 **PEP 302** 的引入模擬 (import emulation)。

pkgutil.**iter_modules**(*path=None*, *prefix=''*)

yield *path* 上所有子模組的*ModuleInfo*，或者如果 *path* 🄵 `None`，則🄵生*sys.path* 上的所有頂層模組。

path 應該是 `None` 或用來尋找模組的路徑串列。

prefix 是在輸出的每個模組名稱前面的輸出字串。

> **備🄵**
>
> 僅適用於有定義 `iter_modules()` 方法的*finder*。此介面🄵非是標准的，因此該模組還提供了*importlib.machinery.FileFinder* 和*zipimport.zipimporter* 的實作。

在 3.3 版的變更: 更新🄵直接基於*importlib*，而不是依賴套件🄵部 **PEP 302** 的引入模擬 (import emulation)。

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

🄵 *path* 上的所有模組遞🄵 yield 出*ModuleInfo*，或如果 *path* 🄵 `None` 則 yield 所有可存取的模組。

path 應該是 `None` 或用來尋找模組的路徑串列。

prefix 是在輸出的每個模組名稱前面的輸出字串。

請注意，此函式必須引入給定之 *path* 上的所有套件（不是所有模組!），以便存取 `__path__` 屬性來尋找子模組。

onerror 是一個函式，如果在嘗試引入套件時發生任何例外，則使用一個引數（正在引入之套件的名稱）來呼叫函式。如果未提供 *onerror* 函式，則會捕獲🄵忽略*ImportError*，同時傳播所有其他例外🄵終止搜尋。

範例:

```
# 列出 Python 可存取的所有模組
walk_packages()

# 列出 ctypes 的所有子模組
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

> **備🄵**
>
> 僅適用於有定義 `iter_modules()` 方法的*finder*。此介面🄵非是標准的，因此該模組還提供了*importlib.machinery.FileFinder* 和*zipimport.zipimporter* 的實作。

在 3.3 版的變更: 更新🄵直接基於*importlib*，而不是依賴套件🄵部 **PEP 302** 的引入模擬 (import emulation)。

`pkgutil.`**`get_data`**(*package*, *resource*)

從套件中取得資源。

這是*loader* *get_data* API 的包裝器。*package* 引數應該是🄵用標准模組格式 (`foo.bar`) 的套件名稱。*resource* 引數應🄵相對檔案名稱的形式，🄵使用 / 作🄵路徑分隔符號。不允許使用父目🄵名稱 `..`，也不允許使用根目🄵名稱（以 / 開頭）。

該函式回傳一個二進位字串，它是指定資源的🄵容。

對於位於檔案系統中且已被引入過的套件，這大致相當於：

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

如果無法定位或載入套件，或者它使用不支援*get_data* 的*loader* 則回傳 `None`。特🄵是*命名空間套件*的*loader* 不支援*get_data*。

`pkgutil.`**`resolve_name`**(*name*)

將名稱解析🄵物件。

標准函式庫中的許多地方都使用了此功能（請參🄵 bpo-12915），且相同功能也被用於擁有廣大使用者的第三方套件，如 setuptools、Django 和 Pyramid。

name 預期要是以下格式之一的字串，其中 W 是有效 Python 識🄵字的簡寫，而點 (dot) 代表這些🄵正規表示式 (pseudo-regex) 中的字面句點 (literal period)：

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

第一種形式僅是🄵了要向後相容。它假設點名稱 (dotted name) 的某些部分是一個套件，其余部分是該套件🄵某處的物件，其可能巢狀地存在於 (nested) 其他物件🄵。由於無法透過檢查 (inspection) 來推斷出套件停止的位置和物件層次結構的開始位置，因此必須使用此形式來重🄵嘗試引入。

在第二種形式中，呼叫者透過使用一個冒號來明確標明分隔點：冒號左側的點名稱是要引入的套件，右側的點名稱是該套件🄵的物件層次結構。這種形式只需要一次引入。如果它以冒號結尾，則回傳一個模組物件。

此函式會回傳一個物件（可能是一個模組），或引發以下其中一個例外：

ValueError -- 如果 *name* 不是可辨識的格式。

ImportError -- 如果在不應該失敗的情🄵下引入失敗。

AttributeError -- 如果在遍歷引入套件中的物件層次結構以取得所需物件時發生失敗。

在 3.9 版被加入.

31.3 `modulefinder` --- 搜尋🄵本所使用的模組

原始碼： Lib/modulefinder.py

此模組提供了一個*ModuleFinder* 類🄵，可用於確定🄵本引入的模組集合。`modulefinder.py` 也可以作🄵🄵本運行，其將 Python 🄵本的檔案名稱作🄵它的引數，🄵在之後會列印出引入模組的報告。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)

記🄵在指定的 *path* 中可以找到名🄵 *pkg_name* 的套件。

modulefinder.**ReplacePackage**(*oldname*, *newname*)

允許指定名🄵 *oldname* 的模組實際上是名🄵 *newname* 的套件。

class modulefinder.**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

此類🄵提供*run_script()* 和*report()* 方法來🄵定🄵本引入的模組集合。*path* 可以是搜尋模組的目🄵串列；如果未指定，則使用 `sys.path`。*debug* 設定偵錯等級；較高的值可使類🄵列印有關其即將執行之操作的偵錯訊息。*excludes* 是要從分析中排除的模組名稱串列。*replace_paths* 是將在模組路徑中替🄵的 `(oldpath, newpath)` 元組串列。

report()

將報告列印到標準輸出，其中列出了🄵本引入的模組及其路徑，以及🄵失或似乎🄵失的模組。

run_script(*pathname*)

分析 *pathname* 檔案的🄵容，該檔案必須包含 Python 程式碼。

modules

將模組名稱對應到模組的字典。請參🄵*ModuleFinder 的用法範例*。

31.3.1 ModuleFinder 的用法范例

將被分析的🄵本 (bacon.py)：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

將輸出 bacon.py 報告的🄵本：

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('載入模組:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('模組🄵有被引入:')
print('\n'.join(finder.badmodules.keys()))
```

範例輸出（可能因架構而🄵）：

```
載入模組:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
```

(繼續下一頁)

(繼續上一頁)

```
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
------------------------------------------------
模組🄵有被引入:
guido.python.ham
baconhameggs
```

31.4 `runpy` --- 定位🄵執行 Python 模組

原始碼： Lib/runpy.py

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a *runpy* function has returned. If that limitation is not acceptable for a given use case, *importlib* is likely to be a more suitable choice than this module.

The *runpy* module provides two functions:

`runpy.`**`run_module`**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module's globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the *__main__* submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by *run_module()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to *True*, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of *sys* is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

> **也參考**
>
> The `-m` option offering equivalent functionality from the command line.

在 3.1 版的變更: Added ability to execute packages by looking for a *__main__* submodule.

在 3.2 版的變更: 新增 `__cached__` 全域變數（請參🄵 **PEP 3147**）。

在 3.4 版的變更: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

在 3.12 版的變更: The setting of `__cached__`, `__loader__`, and `__package__` are deprecated. See *ModuleSpec* for alternatives.

`runpy.`**`run_path`**(*path_name*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module's globals dictionary. As with a script name supplied to the CPython command line, *file_path* may refer to a Python source file, a compiled bytecode file or a valid *sys.path* entry containing a *__main__* module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid *sys.path* entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a *__main__* module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by *run_path()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not *None* and to `'<run_path>'` otherwise.

If *file_path* directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to *file_path*, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to *None*.

If *file_path* is a reference to a valid *sys.path* entry, then `__spec__` will be set appropriately for the imported *__main__* module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the *sys* module. Firstly, *sys.path* may be altered as described above. `sys.argv[0]` is updated with the value of *file_path* and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Note that, unlike *run_module()*, the alterations made to *sys* are not optional in this function as these adjustments are essential to allowing the execution of *sys.path* entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

> **也參考**
>
> using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

在 3.2 版被加入.

在 3.4 版的變更: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid *sys.path* entry rather than being executed directly.

在 3.12 版的變更: The setting of `__cached__`, `__loader__`, and `__package__` are deprecated.

也參考

PEP 338 -- Executing modules as scripts
PEP written and implemented by Nick Coghlan.

PEP 366 -- Main module explicit relative imports
PEP written and implemented by Nick Coghlan.

PEP 451 -- A ModuleSpec Type for the Import System
PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The *importlib.import_module()* function

31.5 `importlib` --- `import` 的實作

在 3.1 版被加入.

原始碼： Lib/importlib/__init__.py

31.5.1 簡介

importlib 的目的可分🄵三個部分。

第一是提供 Python 原始碼中 `import` 陳述式的實作（因此，也延伸到*__import__()* 函式）。這讓 `import` 實作可以移植到任何 Python 直譯器。同時，這也提供了一個比用其他程式語言實作更容易理解的版本。

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Three, the package contains modules exposing additional functionality for managing aspects of Python packages:

- *importlib.metadata* presents access to metadata from third-party distributions.
- *importlib.resources* provides routines for accessing non-code ”resources” from Python packages.

也參考

import
The language reference for the `import` statement.

Packages specification
Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *sys.modules*).

***__import__()* 函式**
The `import` statement is syntactic sugar for this function.

***sys.path* 模組搜尋路徑的初始化**
The initialization of *sys.path*.

PEP 235
Import on Case-Insensitive Platforms

PEP 263
Defining Python Source Code Encodings

PEP 302
New Import Hooks

PEP 328
Imports: Multi-Line and Absolute/Relative

PEP 366
Main module explicit relative imports

PEP 420
Implicit namespace packages

PEP 451
A ModuleSpec Type for the Import System

PEP 488
Elimination of PYO files

PEP 489
多階段擴充模組初始化

PEP 552
Deterministic pycs

PEP 3120
Using UTF-8 as the Default Source Encoding

PEP 3147
PYC Repository Directories

31.5.2 函式

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

An implementation of the built-in *`__import__()`* function.

> **備🄵**
>
> Programmatic importing of modules should use *`import_module()`* instead of this function.

`importlib.import_module`(*name*, *package=None*)

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The *`import_module()`* function acts as a simplifying wrapper around *`importlib.__import__()`*. This means all semantics of the function are derived from *`importlib.__import__()`*. The most important difference between these two functions is that *`import_module()`* returns the specified package or module (e.g. `pkg.mod`), while *`__import__()`* returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call *`invalidate_caches()`* in order for the new module to be noticed by the import system.

在 3.3 版的變更: Parent packages are automatically imported.

`importlib.invalidate_caches`()

Invalidate the internal caches of finders stored at *`sys.meta_path`*. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called

if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

在 3.3 版被加入.

在 3.10 版的變更: Namespace packages created/installed in a different *sys.path* location after the same namespace was already imported are noticed.

importlib.**reload**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in *sys.modules*).

When *reload()* is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects --- with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, *builtins* and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` ... `import` ..., calling *reload()* for the other module does not redefine the objects imported from it --- one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances --- they continue to use the old class definition. The same is true for derived classes.

在 3.4 版被加入.

在 3.7 版的變更: *ModuleNotFoundError* is raised when the module being reloaded lacks a *ModuleSpec*.

⚠ 警告

This function is not thread-safe. Calling it from multiple threads can result in unexpected behavior. It's recommended to use the *threading.Lock* or other synchronization primitives for thread-safe module reloading.

31.5.3 `importlib.abc` -- Abstract base classes related to import

原始碼： Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**MetaPathFinder**

An abstract base class representing a *meta path finder*.

在 3.3 版被加入.

在 3.10 版的變更: No longer a subclass of `Finder`.

find_spec(*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `MetaPathFinders`.

在 3.4 版被加入.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.invalidate_caches()* when invalidating the caches of all finders on *sys.meta_path*.

在 3.4 版的變更: Returns `None` when called instead of *NotImplemented*.

class importlib.abc.**PathEntryFinder**

An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by *importlib.machinery.PathFinder*.

在 3.3 版被加入.

在 3.10 版的變更: No longer a subclass of `Finder`.

find_spec(*fullname*, *target=None*)

An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `PathEntryFinders`.

在 3.4 版被加入.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.machinery.PathFinder.invalidate_caches()* when invalidating the caches of all cached finders.

class importlib.abc.**Loader**

An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader()` method as specified by *importlib.resources.abc.ResourceReader*.

在 3.7 版的變更: Introduced the optional `get_resource_reader()` method.

create_module(*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

在 3.4 版被加入.

在 3.6 版的變更: This method is no longer optional when *exec_module()* is defined.

exec_module(*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when *exec_module()* is called. When this method exists, *create_module()* must be defined.

在 3.4 版被加入.

在 3.6 版的變更: *create_module()* 也必須被定義。

load_module(*fullname*)

A legacy method for loading a module. If the module cannot be loaded, *ImportError* is raised, otherwise the loaded module is returned.

If the requested module already exists in *sys.modules*, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into *sys.modules* before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from *sys.modules*; modules already in *sys.modules* before the loader began execution should be left alone.

The loader should set several attributes on the module (note that some of these attributes can change when a module is reloaded):

- `module.__name__`
- `module.__file__`
- `module.__cached__` （已🄵用）
- `module.__path__`
- `module.__package__` （已🄵用）
- `module.__loader__` （已🄵用）

When *exec_module()* is available then backwards-compatible functionality is provided.

在 3.4 版的變更: Raise *ImportError* when called instead of *NotImplementedError*. Functionality provided when *exec_module()* is available.

Deprecated since version 3.4, will be removed in version 3.15: The recommended API for loading a module is *exec_module()* (and *create_module()*). Loaders should implement it instead of *load_module()*. The import machinery takes care of all the other responsibilities of *load_module()* when *exec_module()* is implemented.

class importlib.abc.**ResourceLoader**

Superseded by TraversableResources

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

在 3.7 版之後被🄵用: This ABC is deprecated in favour of supporting resource loading through *importlib.resources.abc.TraversableResources*. This class exists for backwards compatibility only with other ABCs in this module.

abstractmethod get_data(*path*)

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

在 3.4 版的變更: Raises *OSError* instead of *NotImplementedError*.

class importlib.abc.**InspectLoader**

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code(*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an *ImportError* if loader cannot find the requested module.

> **備🄵**
>
> While the method has a default implementation, it is suggested that it be overridden if possible for performance.

在 3.4 版的變更: No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises *ImportError* if the loader cannot find the module specified.

在 3.4 版的變更: Raises *ImportError* instead of *NotImplementedError*.

is_package(*fullname*)

An optional method to return a true value if the module is a package, a false value otherwise. *ImportError* is raised if the *loader* cannot find the module.

在 3.4 版的變更: Raises *ImportError* instead of *NotImplementedError*.

static source_to_code(*data*, *path='<string>'*)

Create a code object from Python source.

The *data* argument can be whatever the *compile()* function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

在 3.4 版被加入.

在 3.5 版的變更: Made the method static.

exec_module(*module*)

Loader.exec_module() 的實作。

在 3.4 版被加入.

load_module(*fullname*)

Loader.load_module() 的實作。

Deprecated since version 3.4, will be removed in version 3.15: 請改用*exec_module()*。

class importlib.abc.**ExecutionLoader**

An abstract base class which inherits from *InspectLoader* that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename(*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, *ImportError* is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

在 3.4 版的變更: Raises *ImportError* instead of *NotImplementedError*.

class importlib.abc.**FileLoader**(*fullname*, *path*)

An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

在 3.3 版被加入.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module(*fullname*)

Calls super's `load_module()`.

Deprecated since version 3.4, will be removed in version 3.15: Use *Loader.exec_module()* instead.

abstractmethod get_filename(*fullname*)

Returns *path*.

abstractmethod get_data(*path*)

Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.**SourceLoader**

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()***
  Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)

Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

在 3.3 版被加入.

在 3.4 版的變更: Raise *OSError* instead of *NotImplementedError*.

path_mtime(*path*)

Optional abstract method which returns the modification time for the specified path.

在 3.3 版之後被🄵用: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

在 3.4 版的變更: Raise *OSError* instead of *NotImplementedError*.

set_data(*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES*/*PermissionError*), do not propagate the exception.

在 3.4 版的變更: No longer raises *NotImplementedError* when called.

get_code(*fullname*)

Concrete implementation of *InspectLoader.get_code()*.

exec_module(*module*)

Concrete implementation of *Loader.exec_module()*.

在 3.4 版被加入.

load_module(*fullname*)

Concrete implementation of *Loader.load_module()*.

Deprecated since version 3.4, will be removed in version 3.15: Use *exec_module()* instead.

get_source(*fullname*)

Concrete implementation of *InspectLoader.get_source()*.

is_package(*fullname*)

Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

class importlib.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If

the module specified by fullname is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

在 3.7 版被加入.

Deprecated since version 3.12, removed in version 3.14: Use *importlib.resources.abc.TraversableResources* instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, *FileNotFoundError* is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise *FileNotFoundError*.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. *FileNotFoundError* is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.abc.**Traversable**

An object with a subset of *pathlib.Path* methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use *importlib.resources.as_file()*.

在 3.9 版被加入.

Deprecated since version 3.12, removed in version 3.14: Use *importlib.resources.abc.Traversable* instead.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield `Traversable` objects in `self`.

abstractmethod is_dir()

Return `True` if `self` is a directory.

abstractmethod is_file()

Return `True` if `self` is a file.

abstractmethod joinpath(*child*)

Return Traversable child in `self`.

abstractmethod __truediv__(*child*)

Return `Traversable` child in `self`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *pathlib.Path.open*).

When opening as text, accepts encoding parameters such as those accepted by *io.TextIOWrapper*.

read_bytes()

Read contents of `self` as bytes.

read_text(*encoding=None*)

Read contents of `self` as text.

class importlib.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the *importlib.resources.files()* interface. Subclasses *importlib.resources.abc.ResourceReader* and provides concrete implementations of the *importlib.resources.abc.ResourceReader*'s abstract methods. Therefore, any loader supplying *importlib.abc.TraversableResources* also supplies ResourceReader.

Loaders that wish to support resource reading are expected to implement this interface.

在 3.9 版被加入.

Deprecated since version 3.12, removed in version 3.14: Use *importlib.resources.abc.TraversableResources* instead.

abstractmethod files()

Returns a *importlib.resources.abc.Traversable* object for the loaded package.

31.5.4 importlib.machinery -- Importers and path hooks

原始碼： Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

importlib.machinery.**SOURCE_SUFFIXES**

A list of strings representing the recognized file suffixes for source modules.

在 3.3 版被加入.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for non-optimized bytecode modules.

在 3.3 版被加入.

在 3.5 版之後被🄵用: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for optimized bytecode modules.

在 3.3 版被加入.

在 3.5 版之後被🄵用: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**BYTECODE_SUFFIXES**

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

在 3.3 版被加入.

在 3.5 版的變更: The value is no longer dependent on `__debug__`.

importlib.machinery.**EXTENSION_SUFFIXES**

A list of strings representing the recognized file suffixes for extension modules.

在 3.3 版被加入.

importlib.machinery.**all_suffixes**()

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

在 3.3 版被加入.

class importlib.machinery.**BuiltinImporter**

An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

在 3.5 版的變更: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class importlib.machinery.**FrozenImporter**

An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

在 3.4 版的變更: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**

Finder for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

在 3.3 版被加入.

在 3.6 版之後被🄵用: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**

A *Finder* for *sys.path* and package `__path__` attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)

Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

在 3.4 版被加入.

在 3.5 版的變更: If the current working directory -- represented by an empty string -- is no longer valid then `None` is returned but no value is cached in *sys.path_importer_cache*.

classmethod invalidate_caches()

Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to `None` are deleted.

在 3.7 版的變更: Entries of `None` in *sys.path_importer_cache* are deleted.

在 3.4 版的變更: Calls objects in *sys.path_hooks* with the current working directory for `''` (i.e. the empty string).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)

A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call *importlib.invalidate_caches()*.

在 3.3 版被加入.

path

The path the finder will search in.

find_spec(*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within *path*.

在 3.4 版被加入.

invalidate_caches()

Clear out the internal cache.

classmethod path_hook(**loader_details*)

A class method which returns a closure for use on *sys.path_hooks*. An instance of *FileFinder* is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, *ImportError* is raised.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.SourceLoader* by subclassing *importlib.abc.FileLoader* and providing some concrete implementations of other methods.

在 3.3 版被加入.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package(*fullname*)

Return `True` if *path* appears to be for a package.

path_stats(*path*)

Concrete implementation of *importlib.abc.SourceLoader.path_stats()*.

set_data(*path*, *data*)

Concrete implementation of *importlib.abc.SourceLoader.set_data()*.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.FileLoader* which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

在 3.3 版被加入.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package(*fullname*)

Determines if the module is a package based on *path*.

get_code(*fullname*)

Returns the code object for *name* created from *path*.

get_source(*fullname*)

Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)

A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Note that, by default, importing an extension module will fail in subinterpreters if it doesn't implement multi-phase init (see **PEP 489**), even if it would otherwise import successfully.

在 3.3 版被加入.

在 3.12 版的變更: Multi-phase init is now required for use in subinterpreters.

name

Name of the module the loader supports.

path

Path to the extension module.

create_module(*spec*)

Creates the module object from the given specification in accordance with **PEP 489**.

在 3.5 版被加入.

exec_module(*module*)

Initializes the given module object in accordance with **PEP 489**.

在 3.5 版被加入.

is_package(*fullname*)

Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

get_code(*fullname*)

Returns `None` as extension modules lack a code object.

get_source(*fullname*)

Returns `None` as extension modules do not have source code.

get_filename(*fullname*)

Returns *path*.

在 3.4 版被加入.

class importlib.machinery.**NamespaceLoader**(*name*, *path*, *path_finder*)

A concrete implementation of *importlib.abc.InspectLoader* for namespace packages. This is an alias for a private class and is only made public for introspecting the `__loader__` attribute on namespace packages:

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__, importlib.abc.Loader)
True
```

在 3.11 版被加入.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)

A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. Many of these attributes are also available directly on a module: for example, `module.__spec__.origin == module.__file__`. Note, however, that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. For example, it is possible to update the module's `__file__` at runtime and this will not be automatically reflected in the module's *__spec__.origin*, and vice versa.

在 3.4 版被加入.

name

The module's fully qualified name (see `module.__name__`). The *finder* should always set this attribute to a non-empty string.

loader

The *loader* used to load the module (see `module.__loader__`). The *finder* should always set this attribute.

origin

The location the *loader* should use to load the module (see `module.__file__`). For example, for modules loaded from a `.py` file this is the filename. The *finder* should always set this attribute to a meaningful value for the *loader* to use. In the uncommon case that there is not one (like for namespace packages), it should be set to `None`.

submodule_search_locations

A (possibly empty) *sequence* of strings enumerating the locations in which a package's submodules will be found (see `module.__path__`). Most of the time there will only be a single directory in this list.

The *finder* should set this attribute to a sequence, even an empty one, to indicate to the import system that the module is a package. It should be set to `None` for non-package modules. It is set automatically later to a special object for namespace packages.

loader_state

The *finder* may set this attribute to an object containing additional, module-specific data to use when loading the module. Otherwise it should be set to `None`.

cached

The filename of a compiled version of the module's code (see `module.__cached__`). The *finder* should always set this attribute but it may be `None` for modules that do not need compiled code stored.

parent

(Read-only) The fully qualified name of the package the module is in (or the empty string for a top-level module). See `module.__package__`. If the module is a package then this is the same as *name*.

has_location

`True` if the spec's *origin* refers to a loadable location, `False` otherwise. This value impacts how `origin` is interpreted and how the module's `__file__` is populated.

class `importlib.machinery.`**AppleFrameworkLoader**(*name*, *path*)

A specialization of *importlib.machinery.ExtensionFileLoader* that is able to load extension modules in Framework format.

For compatibility with the iOS App Store, *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the `Frameworks` folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

To accommodate this requirement, when running on iOS, extension module binaries are *not* packaged as `.so` files on `sys.path`, but as individual standalone frameworks. To discover those frameworks, this loader is be registered against the `.fwork` file extension, with a `.fwork` file acting as a placeholder in the original location of the binary on `sys.path`. The `.fwork` file contains the path of the actual binary in the `Frameworks` folder, relative to the app bundle. To allow for resolving a framework-packaged binary back to the original location, the framework is expected to contain a `.origin` file that contains the location of the `.fwork` file, relative to the app bundle.

For example, consider the case of an import `from foo.bar import _whiz`, where `_whiz` is implemented with the binary module `sources/foo/bar/_whiz.abi3.so`, with `sources` being the location registered on `sys.path`, relative to the application bundle. This module *must* be distributed as `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (creating the framework name from the full import path of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When a module is loaded with this loader, the `__file__` for the module will report as the location of the `.fwork` file. This allows code to use the `__file__` of a module as an anchor for file system traversal. However, the spec origin will reference the location of the *actual* binary in the `.framework` folder.

The Xcode project building the app is responsible for converting any `.so` files from wherever they exist in the `PYTHONPATH` into frameworks in the `Frameworks` folder (including stripping extensions from the module file, the addition of framework metadata, and signing the resulting framework), and creating the `.fwork` and `.origin` files. This will usually be done with a build step in the Xcode project; see the iOS documentation for details on how to construct this build step.

在 3.13 版被加入.

可用性: iOS.

name

Name of the module the loader supports.

path

Path to the `.fwork` file for the extension module.

31.5.5 `importlib.util` -- Utility code for importers

原始碼： Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

importlib.util.**MAGIC_NUMBER**

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *importlib.abc.SourceLoader*.

在 3.4 版被加入.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)

Return the **PEP 3147/PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpreter's optimization level to be used. Any other value's string representation is used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *TypeError* is raised.

在 3.4 版被加入.

在 3.5 版的變更: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

在 3.6 版的變更: Accepts a *path-like object*.

importlib.util.**source_from_cache**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

在 3.4 版被加入.

在 3.6 版的變更: Accepts a *path-like object*.

importlib.util.**decode_source**(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

在 3.4 版被加入.

importlib.util.**resolve_name**(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __spec__.parent)` without doing a check to see if the **package** argument is needed.

ImportError is raised if **name** is a relative module name but **package** is a false value (e.g. `None` or the empty string). *ImportError* is also raised if a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

在 3.3 版被加入.

在 3.9 版的變更: To improve consistency with import statements, raise *ImportError* instead of *ValueError* for invalid relative import attempts.

importlib.util.**find_spec**(*name*, *package=None*)

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

在 3.4 版被加入.

在 3.7 版的變更: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

`importlib.util.`**`module_from_spec`**(*spec*)

Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return `None`, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

在 3.5 版被加入.

`importlib.util.`**`spec_from_loader`**(*name*, *loader*, ***, *origin=None*, *is_package=None*)

A factory function for creating a *ModuleSpec* instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

在 3.4 版被加入.

`importlib.util.`**`spec_from_file_location`**(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a *ModuleSpec* instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

在 3.4 版被加入.

在 3.6 版的變更: Accepts a *path-like object*.

`importlib.util.`**`source_hash`**(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the *source_hash()* of the corresponding source file's contents in its header.

在 3.7 版被加入.

`importlib.util.`**`_incompatible_extension_module_restrictions`**(***, *disable_check*)

A context manager that can temporarily skip the compatibility check for extension modules. By default the check is enabled and will fail when a single-phase init module is imported in a subinterpreter. It will also fail for a multi-phase init module that doesn't explicitly support a per-interpreter GIL, when imported in an interpreter with its own GIL.

Note that this function is meant to accommodate an unusual case; one which is likely to eventually go away. There's is a pretty good chance this is not what you were looking for.

你可以透過實作多階段初始化的基本介面 (**PEP 489**) 🄵假稱有多個直譯器（或直譯器各🄵持有的 GIL）的支援，來達到與此函式相同的效果。

警告

Using this function to disable the check can lead to unexpected behavior and even crashes. It should only be used during extension module development.

在 3.12 版被加入.

class importlib.util.**LazyLoader**(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

> **備🄵**
>
> For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

在 3.5 版被加入.

在 3.6 版的變更: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod factory(*loader*)

A class method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6 范例

Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

Note that if `name` is a submodule (contains a dot), *importlib.util.find_spec()* will import the parent module.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
```

(繼續下一頁)

(繼續上一頁)

```
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

Importing a source file directly

This recipe should be used with caution: it is an approximation of an import statement where the file path is specified directly, rather than *sys.path* being searched. Alternatives should first be considered first, such as modifying *sys.path* when a proper module is required, or using *runpy.run_path()* when the global namespace resulting from running a Python file is appropriate.

To import a Python source file directly from a path, use the following recipe:

```
import importlib.util
import sys


def import_from_path(module_name, file_path):
    spec = importlib.util.spec_from_file_location(module_name, file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module


# For illustrative purposes only (use of `json` is arbitrary).
import json
file_path = json.__file__
module_name = json.__name__

# Similar outcome as `import json`.
json = import_from_path(module_name, file_path)
```

Implementing lazy imports

The example below shows how to implement lazy imports:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
>>> #but it is not loaded in memory yet.
>>> lazy_typing.TYPE_CHECKING
False
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on *sys.meta_path* while the latter is what you create using a *path entry hook* on *sys.path_hooks* which works with *sys.path* entries to potentially create

a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *`importlib.import_module()`*:

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 `importlib.resources` -- 套件資源的讀取、開🄵與存取

原始碼： Lib/importlib/resources/__init__.py

在 3.7 版被加入.

This module leverages Python's import system to provide access to *resources* within *packages*.

"Resources" are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, Python module sources (.py) of a package and compilation artifacts (pycache) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system: for example, a package and its resources can be imported from a zip file using *zipimport*.

備🄵

This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.resources.abc.ResourceReader*.

class importlib.resources.**Anchor**

Represents an anchor for resources, either a *module object* or a module name as a string. Defined as `Union[str, ModuleType]`.

importlib.resources.**files**(*anchor:* Anchor | None = *None*)

Returns a *Traversable* object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional *Anchor*. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller's module is used.

在 3.9 版被加入.

在 3.12 版的變更: *package* parameter was renamed to *anchor*. *anchor* can now be a non-package module and if omitted will default to the caller's module. *package* is still accepted for compatibility but will raise a *DeprecationWarning*. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

importlib.resources.**as_file**(*traversable*)

Given a *Traversable* object representing a file or directory, typically from *importlib.resources.files()*, return a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

在 3.9 版被加入.

在 3.12 版的變更: Added support for *traversable* representing a directory.

31.6.1 Functional API

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions:

- *anchor* is an `Anchor`, as in `files()`. Unlike in `files`, it may not be omitted.
- *path_names* are components of a resource's path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use:

```
importlib.resources.read_text(my_module, "info.txt")
```

  Like `Traversable.joinpath`, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent:

```
importlib.resources.read_binary(my_module, "pics/painting.png")
importlib.resources.read_binary(my_module, "pics", "painting.png")
```

  For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use:

```
importlib.resources.read_text(my_module, "info", "chapter1.txt",
                              encoding='utf-8')
```

`importlib.resources.`**`open_binary`**(*anchor*, **path_names*)

Open the named resource for binary reading.

See *the introduction* for details on *anchor* and *path_names*.

This function returns a `BinaryIO` object, that is, a binary stream open for reading.

這個函式大致等同於:

```
files(anchor).joinpath(*path_names).open('rb')
```

在 3.13 版的變更: 多個 *path_names* 是被接受的。

`importlib.resources.`**`open_text`**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open()`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a `TextIO` object, that is, a text stream open for reading.

這個函式大致等同於:

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

在 3.13 版的變更: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.`**`read_binary`**(*anchor*, **path_names*)

以`bytes` 形式讀取🄵回傳指定資源的🄵容。

See *the introduction* for details on *anchor* and *path_names*.

這個函式大致等同於:

```
files(anchor).joinpath(*path_names).read_bytes()
```

在 3.13 版的變更: 多個 *path_names* 是被接受的。

importlib.resources.**read_text**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Read and return the contents of the named resource as *str*. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in *open()*.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

這個函式大致等同於:

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

在 3.13 版的變更: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.**path**(*anchor*, **path_names*)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the *stat()* method requires an actual file system path; it can be used like this:

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

See *the introduction* for details on *anchor* and *path_names*.

這個函式大致等同於:

```
as_file(files(anchor).joinpath(*path_names))
```

在 3.13 版的變更: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.**is_resource**(*anchor*, **path_names*)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See *the introduction* for details on *anchor* and *path_names*.

這個函式大致等同於:

```
files(anchor).joinpath(*path_names).is_file()
```

在 3.13 版的變更: 多個 *path_names* 是被接受的。

importlib.resources.**contents**(*anchor*, **path_names*)

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as *str*. The iterable does not recurse into subdirectories.

See *the introduction* for details on *anchor* and *path_names*.

這個函式大致等同於:

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

在 3.11 版之後被🄵用: Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

31.7 importlib.resources.abc -- 資源的抽象基底類🄵

原始碼： Lib/importlib/resources/abc.py

在 3.11 版被加入.

class importlib.resources.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

在 3.12 版之後被🄵用: Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, `FileNotFoundError` is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise `FileNotFoundError`.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. `FileNotFoundError` is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.resources.abc.**Traversable**

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield Traversable objects in self.

abstractmethod is_dir()

Return `True` if self is a directory.

abstractmethod is_file()

Return `True` if self is a file.

abstractmethod joinpath(**pathsegments*)

Traverse directories according to *pathsegments* and return the result as `Traversable`.

Each *pathsegments* argument may contain multiple names separated by forward slashes (`/`, `posixpath.sep`). For example, the following are equivalent:

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

Note that some `Traversable` implementations might not be updated to the latest version of the protocol. For compatibility with such implementations, provide a single argument without path separators to each call to `joinpath`. For example:

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('file.txt')
```

在 3.11 版的變更: `joinpath` accepts multiple *pathsegments*, and these segments may contain forward slashes as path separators. Previously, only a single *child* argument was accepted.

abstractmethod __truediv__(*child*)

Return Traversable child in self. Equivalent to `joinpath(child)`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *`pathlib.Path.open`*).

When opening as text, accepts encoding parameters such as those accepted by *`io.TextIOWrapper`*.

read_bytes()

Read contents of self as bytes.

read_text(*encoding=None*)

Read contents of self as text.

class importlib.resources.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the *`importlib.resources.files()`* interface. Subclasses *`ResourceReader`* and provides concrete implementations of the `ResourceReader`'s abstract methods. Therefore, any loader supplying `TraversableResources` also supplies `ResourceReader`.

Loaders that wish to support resource reading are expected to implement this interface.

abstractmethod files()

Returns a *`importlib.resources.abc.Traversable`* object for the loaded package.

31.8 `importlib.metadata` -- 存取套件的元資料

在 3.8 版被加入.

在 3.10 版的變更: `importlib.metadata` is no longer provisional.

原始碼： Lib/importlib/metadata/__init__.py

`importlib.metadata` is a library that provides access to the metadata of an installed Distribution Package, such as its entry points or its top-level names (Import Packages, modules, if any). Built in part on Python's import system, this library intends to replace similar functionality in the entry point API and metadata API of `pkg_resources`.

Along with *importlib.resources*, this package can eliminate the need to use the older and less efficient `pkg_resources` package.

`importlib.metadata` operates on third-party *distribution packages* installed into Python's `site-packages` directory via tools such as pip. Specifically, it works with distributions with discoverable `dist-info` or `egg-info` directories, and metadata defined by the Core metadata specifications.

重要

These are *not* necessarily equivalent to or correspond 1:1 with the top-level *import package* names that can be imported inside Python code. One *distribution package* can contain multiple *import packages* (and single modules), and one top-level *import package* may map to multiple *distribution packages* if it is a namespace package. You can use *packages_distributions()* to get a mapping between them.

By default, distribution metadata can live on the file system or in zip archives on *sys.path*. Through an extension mechanism, the metadata can live almost anywhere.

也參考

https://importlib-metadata.readthedocs.io/
The documentation for `importlib_metadata`, which supplies a backport of `importlib.metadata`. This includes an API reference for this module's classes and functions, as well as a migration guide for existing users of `pkg_resources`.

31.8.1 Overview

Let's say you wanted to get the version string for a Distribution Package you've installed using `pip`. We start by creating a virtual environment and installing something into it:

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

You can get the version string for `wheel` by running the following:

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get a collection of entry points selectable by properties of the EntryPoint (typically 'group' or 'name'), such as `console_scripts`, `distutils.commands` and others. Each group contains a collection of *EntryPoint* objects.

You can get the *metadata for a distribution*:

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-email',
↪'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL', 'Project-URL',
↪'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier
↪', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Requires-Python', 'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

You can also get a *distribution's version number*, list its *constituent files*, and get a list of the distribution's *Distribution requirements*.

exception importlib.metadata.**PackageNotFoundError**

Subclass of *ModuleNotFoundError* raised by several functions in this module when queried for a distribution package which is not installed in the current Python environment.

31.8.2 Functional API

This package provides the following functionality via its public API.

Entry points

`importlib.metadata.`**`entry_points`**(***select_params*)

Returns a *`EntryPoints`* instance describing entry points for the current environment. Any given keyword parameters are passed to the `select()` method for comparison to the attributes of the individual entry point definitions.

Note: it is not currently possible to query for entry points based on their `EntryPoint.dist` attribute (as different `Distribution` instances do not currently compare equal, even if they have the same attributes)

`class` `importlib.metadata.`**`EntryPoints`**

Details of a collection of installed entry points.

Also provides a `.groups` attribute that reports all identified entry point groups, and a `.names` attribute that reports all identified entry point names.

`class` `importlib.metadata.`**`EntryPoint`**

Details of an installed entry point.

Each `EntryPoint` instance has `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value. There are also `.module`, `.attr`, and `.extras` attributes for getting the components of the `.value` attribute, and `.dist` for obtaining information regarding the distribution package that provides the entry point.

Query all entry points:

```
>>> eps = entry_points()
```

The `entry_points()` function returns a `EntryPoints` object, a collection of all `EntryPoint` objects with `names` and `groups` attributes for convenience:

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.writers',
↪'setuptools.installation']
```

`EntryPoints` has a `select()` method to select entry points matching specific properties. Select entry points in the `console_scripts` group:

```
>>> scripts = eps.select(group='console_scripts')
```

Equivalently, since `entry_points()` passes keyword arguments through to select:

```
>>> scripts = entry_points(group='console_scripts')
```

Pick out a specific script named "wheel" (found in the wheel project):

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

Equivalently, query for that entry point during selection:

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

Inspect the resolved entry point:

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
```

(繼續下一頁)

(繼續上一頁)

```
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entry points, their definition, and usage.

在 3.12 版的變更: The "selectable" entry points were introduced in `importlib_metadata` 3.6 and Python 3.10. Prior to those changes, `entry_points` accepted no parameters and always returned a dictionary of entry points, keyed by group. With `importlib_metadata` 5.0 and Python 3.12, `entry_points` always returns an `EntryPoints` object. See backports.entry_points_selectable for compatibility options.

在 3.13 版的變更: `EntryPoint` objects no longer present a tuple-like interface (`__getitem__()`).

Distribution metadata

`importlib.metadata.`**`metadata`**(*distribution_name*)

Return the distribution metadata corresponding to the named distribution package as a *`PackageMetadata`* instance.

Raises *`PackageNotFoundError`* if the named distribution package is not installed in the current Python environment.

`class` `importlib.metadata.`**`PackageMetadata`**

A concrete implementation of the PackageMetadata protocol.

In addition to providing the defined protocol methods and attributes, subscripting the instance is equivalent to calling the `get()` method.

Every Distribution Package includes some metadata, which you can extract using the `metadata()` function:

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure name the metadata keywords, and the values are returned unparsed from the distribution metadata:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

`PackageMetadata` also presents a `json` attribute that returns all the metadata in a JSON-compatible form per **PEP 566**:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

在 3.10 版的變更: The `Description` is now included in the metadata when presented through the payload. Line continuation characters have been removed.

新增 `json` 屬性。

Distribution versions

`importlib.metadata.`**`version`**(*distribution_name*)

Return the installed distribution package version for the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

The `version()` function is the quickest way to get a Distribution Package's version number, as a string:

```
>>> version('wheel')
'0.32.3'
```

Distribution files

`importlib.metadata.`**`files`**(*distribution_name*)

Return the full set of files contained within the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

Returns *None* if the distribution is found but the installation database records reporting the files associated with the distribution package are missing.

`class` `importlib.metadata.`**`PackagePath`**

A *pathlib.PurePath* derived object with additional `dist`, `size`, and `hash` properties corresponding to the distribution package's installation metadata for that file.

The `files()` function takes a Distribution Package name and returns all of the files installed by this distribution. Each file is reported as a *PackagePath* instance. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

Once you have the file, you can also read its contents:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

You can also use the `locate()` method to get the absolute path to the file:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

In the case where the metadata file listing files (`RECORD` or `SOURCES.txt`) is missing, `files()` will return *None*. The caller may wish to wrap calls to `files()` in always_iterable or otherwise guard against this condition if the target distribution is not known to have the metadata present.

Distribution requirements

`importlib.metadata.`**`requires`**(*distribution_name*)

Return the declared dependency specifiers for the named distribution package.

Raises *`PackageNotFoundError`* if the named distribution package is not installed in the current Python environment.

To get the full set of requirements for a Distribution Package, use the `requires()` function:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

Mapping import to distribution packages

`importlib.metadata.`**`packages_distributions`**()

Return a mapping from the top level module and import package names found via *`sys.meta_path`* to the names of the distribution packages (if any) that provide the corresponding files.

To allow for namespace packages (which may have members provided by multiple distribution packages), each top level import name maps to a list of distribution names rather than mapping directly to a single name.

A convenience method to resolve the Distribution Package name (or names, in the case of a namespace package) that provide each importable top-level Python module or Import Package:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': ['jaraco.classes
↪', 'jaraco.functools'], ...}
```

Some editable installs, do not supply top-level names, and thus this function is not reliable with such installs.

在 3.10 版被加入.

31.8.3 Distributions

`importlib.metadata.`**`distribution`**(*distribution_name*)

Return a *`Distribution`* instance describing the named distribution package.

Raises *`PackageNotFoundError`* if the named distribution package is not installed in the current Python environment.

`class` `importlib.metadata.`**`Distribution`**

Details of an installed distribution package.

Note: different `Distribution` instances do not currently compare equal, even if they relate to the same installed distribution and accordingly have the same attributes.

While the module level API described above is the most common and convenient usage, you can get all of that information from the `Distribution` class. `Distribution` is an abstract object that represents the metadata for a Python Distribution Package. You can get the concrete `Distribution` subclass instance for an installed distribution package by calling the *`distribution()`* function:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
>>> type(dist)
<class 'importlib.metadata.PathDistribution'>
```

Thus, an alternative way to get the version number is through the `Distribution` instance:

```
>>> dist.version
'0.32.3'
```

There are all kinds of additional metadata available on `Distribution` instances:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

For editable packages, an `origin` property may present **PEP 610** metadata:

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

在 3.13 版被加入: 新增 `.origin` 屬性 (property)。

31.8.4 Distribution Discovery

By default, this package provides built-in support for discovery of metadata for file system and zip file Distribution Packages. This metadata finder search defaults to `sys.path`, but varies slightly in how it interprets those values from how other import machinery does. In particular:

- `importlib.metadata` does not honor *`bytes`* objects on `sys.path`.
- `importlib.metadata` will incidentally honor *`pathlib.Path`* objects on `sys.path` even though such values will be ignored for imports.

31.8.5 Implementing Custom Providers

`importlib.metadata` address two API surfaces, one for *consumers* and another for *providers*. Most users are consumers, consuming metadata provided by the packages. There are other use-cases, however, where users wish to expose metadata through some other mechanism, such as alongside a custom importer. Such a use case calls for a *custom provider*.

Because Distribution Package metadata is not available through *`sys.path`* searches, or package loaders directly, the metadata for a distribution is found through import system finders. To find a distribution package's metadata, `importlib.metadata` queries the list of *meta path finders* on *`sys.meta_path`*.

The implementation has hooks integrated into the `PathFinder`, serving metadata for distribution packages found on the file system.

The abstract class *`importlib.abc.MetaPathFinder`* defines the interface expected of finders by Python's import system. `importlib.metadata` extends this protocol by looking for an optional `find_distributions` callable on the finders from *`sys.meta_path`* and presents this extended interface as the `DistributionFinder` abstract base class, which defines this abstract method:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()) -> Iterable[Distribution]:
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and name to match and may supply other relevant context sought by the consumer.

In practice, to support finding distribution package metadata in locations other than the file system, subclass `Distribution` and implement the abstract methods. Then from a custom finder, return instances of this derived `Distribution` in the `find_distributions()` method.

范例

Imagine a custom finder that loads Python modules from a database:

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

That importer now presumably provides importable modules from a database, but it provides no metadata or entry points. For this custom importer to provide metadata, it would also need to implement `DistributionFinder`:

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

In this way, `query_distributions` would return records for each distribution served by the database matching the query. For example, if `requests-1.0` is in the database, `find_distributions` would yield a `DatabaseDistribution` for `Context(name='requests')` or `Context(name=None)`.

For the sake of simplicity, this example ignores `context.path`. The `path` attribute defaults to `sys.path` and is the set of import paths to be considered in the search. A `DatabaseImporter` could potentially function without any concern for a search path. Assuming the importer does no partitioning, the "path" would be irrelevant. In order to illustrate the purpose of `path`, the example would need to illustrate a more complex `DatabaseImporter` whose behavior varied depending on `sys.path`/`PYTHONPATH`. In that case, the `find_distributions` should honor the `context.path` and only yield `Distribution`s pertinent to that path.

`DatabaseDistribution`, then, would look something like:

```
class DatabaseDistribution(importlib.metadata.Distribution):
    def __init__(self, record):
        self.record = record

    def read_text(self, filename):
        """
        Read a file like "METADATA" for the current distribution.
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
              for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file system")
```

This basic implementation should provide metadata and entry points for packages served by the `DatabaseImporter`, assuming that the `record` supplies suitable `.name`, `.version`, and `.entry_points` attributes.

The `DatabaseDistribution` may also provide other metadata files, like `RECORD` (required for `Distribution.files`) or override the implementation of `Distribution.files`. See the source for more inspiration.

31.9 `sys.path` 模組搜尋路徑的初始化

A module search path is initialized when Python starts. This module search path may be accessed at *`sys.path`*.

The first entry in the module search path is the directory that contains the input script, if there is one. Otherwise, the first entry is the current directory, which is the case when executing the interactive shell, a `-c` command, or `-m` module.

The `PYTHONPATH` environment variable is often used to add directories to the search path. If this environment variable is found then the contents are added to the module search path.

> **備🄵**
>
> `PYTHONPATH` will affect all installed Python versions/environments. Be wary of setting this in your shell profile or global environment variables. The *`site`* module offers more nuanced techniques as mentioned below.

The next items added are the directories containing standard Python modules as well as any *extension module*s that these modules depend on. Extension modules are `.pyd` files on Windows and `.so` files on other platforms. The directory with the platform-independent Python modules is called `prefix`. The directory with the extension modules is called `exec_prefix`.

The `PYTHONHOME` environment variable may be used to set the `prefix` and `exec_prefix` locations. Otherwise these directories are found by using the Python executable as a starting point and then looking for various 'landmark' files and directories. Note that any symbolic links are followed so the real Python executable location is used as the search starting point. The Python executable location is called `home`.

Once `home` is determined, the `prefix` directory is found by first looking for `python`*`majorversionminorversion`*`.zip` (`python311.zip`). On Windows the zip archive is searched for in `home` and on Unix the archive is expected to be in `lib`. Note that the expected zip archive location is added to the module search path even if the archive does not exist. If no archive was found, Python on Windows will continue the search for `prefix` by looking for `Lib\os.py`. Python on Unix will look for `lib/python`*`majorversion`*`.`*`minorversion`*`/os.py` (`lib/python3.11/os.py`). On Windows `prefix` and `exec_prefix` are the same, however on other platforms `lib/python`*`majorversion`*`.`*`minorversion`*`/lib-dynload` (`lib/python3.11/lib-dynload`) is searched for and used as an anchor for `exec_prefix`. On some platforms `lib` may be `lib64` or another value, see *`sys.platlibdir`* and `PYTHONPLATLIBDIR`.

Once found, `prefix` and `exec_prefix` are available at *`sys.base_prefix`* and *`sys.base_exec_prefix`* respectively.

If `PYTHONHOME` is not set, and a `pyvenv.cfg` file is found alongside the main executable, or in its parent directory, *`sys.prefix`* and *`sys.exec_prefix`* get set to the directory containing `pyvenv.cfg`, otherwise they are set to the same value as *`sys.base_prefix`* and *`sys.base_exec_prefix`*, respectively. This is used by 🄵擬環境.

Finally, the *`site`* module is processed and `site-packages` directories are added to the module search path. A common way to customize the search path is to create *`sitecustomize`* or *`usercustomize`* modules as described in the *`site`* module documentation.

> **備🄵**
>
> Certain command line options may further affect path calculations. See `-E`, `-I`, `-s` and `-S` for further details.

在 3.14 版的變更: *`sys.prefix`* and *`sys.exec_prefix`* are now set to the `pyvenv.cfg` directory during the path initialization. This was previously done by *`site`*, therefore affected by `-S`.

31.9.1 🄵擬環境

Virtual environments place a `pyvenv.cfg` file in their prefix, which causes *`sys.prefix`* and *`sys.exec_prefix`* to point to them, instead of the base installation.

The `prefix` and `exec_prefix` values of the base installation are available at *`sys.base_prefix`* and *`sys.base_exec_prefix`*.

As well as being used as a marker to identify virtual environments, `pyvenv.cfg` may also be used to configure the *`site`* initialization. Please refer to *`site`*'s *virtual environments documentation*.

備🄵

`PYTHONHOME` overrides the `pyvenv.cfg` detection.

備🄵

There are other ways how "virtual environments" could be implemented, this documentation refers implementations based on the `pyvenv.cfg` mechanism, such as *`venv`*. Most virtual environment implementations follow the model set by *`venv`*, but there may be exotic implementations that diverge from it.

31.9.2 _pth 檔案

To completely override *`sys.path`* create a `._pth` file with the same name as the shared library or executable (`python._pth` or `python311._pth`). The shared library path is always known on Windows, however it may not be available on other platforms. In the `._pth` file specify one line for each path to add to *`sys.path`*. The file based on the shared library name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and *`site`* is not imported unless one line in the file specifies `import site`. Blank paths and lines starting with `#` are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to `site` are not permitted, and arbitrary code cannot be specified.

Note that `.pth` files (without leading underscore) will be processed normally by the *`site`* module when `import site` has been specified.

31.9.3 Embedded Python

If Python is embedded within another application `Py_InitializeFromConfig()` and the `PyConfig` structure can be used to initialize Python. The path specific details are described at init-path-config.

也參考

- windows_finding_modules 有關於 Windows 的詳細資訊。
- using-on-unix 有關於 Unix 的詳細資訊。

CHAPTER 32

Python 語言服務

Python 提供了許多模組來協助處理 Python 語言。這些模組支援標記化、剖析、語法分析、位元組碼反組譯，以及其他各種設施。

這些模組包括：

32.1 `ast` --- 抽象語法樹 (Abstract Syntax Trees)

原始碼： Lib/ast.py

---

ast 模組可以幫助 Python 應用程式處理 Python 抽象語法文法 (abstract syntax grammar) 樹狀資料結構。抽象語法本身可能會隨著每個 Python 版本發布而改變；此模組有助於以程式化的方式來得知目前文法的面貌。

要生成抽象語法樹，可以透過將*ast.PyCF_ONLY_AST* 作🄵旗標傳遞給🄵建函式*compile()* 或使用此模組所提供的*parse()* 輔助函式。結果將會是一個物件的樹，其類🄵都繼承自*ast.AST*。可以使用🄵建的*compile()* 函式將抽象語法樹編譯成 Python 程式碼物件。

32.1.1 抽象文法 (Abstract Grammar)

抽象文法目前定義如下：

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment, type_param* type_params)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
```

(繼續下一頁)

(繼續上一頁)

```
                          string? type_comment, type_param* type_params)

        | ClassDef(identifier name,
           expr* bases,
           keyword* keywords,
           stmt* body,
           expr* decorator_list,
           type_param* type_params)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, expr value, string? type_comment)
        | TypeAlias(expr name, type_param* type_params, expr value)
        | AugAssign(expr target, operator op, expr value)
        -- 'simple' indicates that we annotate simple name without parens
        | AnnAssign(expr target, expr annotation, expr? value, int simple)

        -- use 'orelse' because else is a keyword in target languages
        | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
        | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
        | While(expr test, stmt* body, stmt* orelse)
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body, string? type_comment)
        | AsyncWith(withitem* items, stmt* body, string? type_comment)

        | Match(expr subject, match_case* cases)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
        | Nonlocal(identifier* names)
        | Expr(expr value)
        | Pass | Break | Continue

        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

        -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr?* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
```

(繼續下一頁)

(繼續上一頁)

```
        -- x < 4 < 3 and (x < 4) < 3
        | Compare(expr left, cmpop* ops, expr* comparators)
        | Call(expr func, expr* args, keyword* keywords)
        | FormattedValue(expr value, int conversion, expr? format_spec)
        | Interpolation(expr value, constant str, int conversion, expr? format_spec)
        | JoinedStr(expr* values)
        | TemplateStr(expr* values)
        | Constant(constant value, string? kind)

        -- the following expression can appear in assignment context
        | Attribute(expr value, identifier attr, expr_context ctx)
        | Subscript(expr value, expr slice, expr_context ctx)
        | Starred(expr value, expr_context ctx)
        | Name(identifier id, expr_context ctx)
        | List(expr* elts, expr_context ctx)
        | Tuple(expr* elts, expr_context ctx)

        -- can appear only in Subscript
        | Slice(expr? lower, expr? upper, expr? step)

         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    expr_context = Load | Store | Del

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
              attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
```

(繼續下一頁)

(繼續上一頁)

```
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs, pattern* kwd_
↪patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra mapping␣
↪keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? default_value)
               | ParamSpec(identifier name, expr? default_value)
               | TypeVarTuple(identifier name, expr? default_value)
               attributes (int lineno, int col_offset, int end_lineno, int end_col_offset)
}
```

32.1.2 節點 (Node) 類🄵

class ast.**AST**

這是所有 AST 節點類🄵的基礎。實際的節點類🄵是衍生自 `Parser/Python.asdl` 檔案，該檔案在上方 重現。它們被定義於 `_ast` 的 C 模組中，🄵於*ast* 中重新匯出。

抽象文法中🄵每個左側符號定義了一個類🄵（例如 `ast.stmt` 或 `ast.expr`）。此外，也🄵每個右側的建構函式 (constructor) 定義了一個類🄵；這些類🄵繼承自左側樹的類🄵。例如，*ast.BinOp* 繼承自 `ast.expr`。對於具有替代方案（即🄵「和 (sums)」）的生🄵規則，左側類🄵是抽象的：僅有特定建構函式節點的實例會被建立。

_fields

每個具體類🄵都有一個屬性 `_fields`，它會給出所有子節點的名稱。

具體類🄵的每個實例對於每個子節點都有一個屬性，其型🄵如文法中所定義。例如，*ast.BinOp* 實例具有型🄵🄵 `ast.expr` 的屬性 `left`。

如果這些屬性在文法中被標記🄵可選（使用問號），則該值可能🄵 `None`。如果屬性可以有零個或多個值（用星號標記），則這些值將表示🄵 Python 串列。使用*compile()* 編譯 AST 時，所有可能的屬性都必須存在🄵且具有有效值。

_field_types

每個具體類🄵上的 `_field_types` 屬性是將欄位名稱（也在*_fields* 中列出）對映到其型🄵的字典。

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None, 'default_value': ast.expr | None}
```

在 3.13 版被加入.

lineno

col_offset

end_lineno

end_col_offset

`ast.expr` 和 `ast.stmt` 子類🄵的實例具有*lineno*、*col_offset*、*end_lineno* 和*end_col_offset* 屬性。*lineno* 和*end_lineno* 是原始文本跨度 (source text span) 的第一個和最後一個列號（1-indexed，因此第一列號是 1）以及*col_offset* 和*end_col_offset* 是生成節點的第一個和最後一個標記對應的 UTF-8 位元組偏移量。會記🄵 UTF-8 偏移量是因🄵剖析器 (parser) 🄵部使用 UTF-8。

請注意，編譯器🄵不需要結束位置，因此其🄵可選的。結束偏移量在最後一個符號之後，例如可以使用 `source_line[node.col_offset : node.end_col_offset]` 來取得單列運算式節點 (expression node) 的原始片段。

`ast.T` 類🄵的建構函式按以下方式剖析其引數：

- 如果有位置引數，則必須與 `T._fields` 中的項目一樣多；它們將被賦🄵🄵這些名稱的屬性。
- 如果有關鍵字引數，它們會將相同名稱的屬性設定🄵給定值。

例如，要建立🄵填充 (populate) *ast.UnaryOp* 節點，你可以使用：

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

如果建構函式中省略了文法中可選的欄位，則它預設🄵 `None`。如果省略串列欄位，則預設🄵空串列。如果省略 `ast.expr_context` 型🄵的欄位，則預設🄵*Load()*。如果省略任何其他欄位，則會引發*DeprecationWarning*，且 AST 節點將🄵有此欄位。在 Python 3.15 中，這種情🄵會引發錯誤。

在 3.8 版的變更: *ast.Constant* 類🄵現在用於所有常數。

在 3.9 版的變更: 以它們的值表示簡單索引，擴充切片 (slice) 則以元組 (tuple) 表示。

在 3.14 版的變更: The `__repr__()` output of *AST* nodes includes the values of the node fields.

在 3.8 版之後被🄵用: 舊的類🄵 `ast.Num`、`ast.Str`、`ast.Bytes`、`ast.NameConstant` 和 `ast.Ellipsis` 仍然可用，但它們將在未來的 Python 釋出版本中移除。與此同時，實例化它們將回傳不同類🄵的實例。

在 3.9 版之後被🄵用: 舊的類🄵 `ast.Index` 和 `ast.ExtSlice` 仍然可用，但它們將在未來的 Python 版本中🄵除。同時，實例化它們會回傳不同類🄵的實例。

Deprecated since version 3.13, will be removed in version 3.15: 先前版本的 Python 允許建立缺少必填欄位的 AST 節點。同樣地，AST 節點建構函式允許將任意關鍵字引數設🄵 AST 節點的屬性，即使它們與 AST 節點的任何欄位都不匹配。此行🄵已被🄵用，🄵將在 Python 3.15 中🄵除。

備🄵

這🄵顯示的特定節點類🄵的描述最初是從出色的 Green Tree Snakes 專案和所有貢獻者那🄵改編而來的。

根節點

class `ast.`**Module**(*body*, *type_ignores*)

一個 Python 模組，與檔案輸入一樣。由*ast.parse()* 在預設的 `"exec"` *mode* 下生成的節點型🄵。

`body` 是模組的陳述式 的一個*list*。

`type_ignores` 是模組的忽略型🄵🄵解的*list*；有關更多詳細資訊，請參🄵*ast.parse()*。

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class `ast.`**Expression**(*body*)

單個 Python 運算式輸入。當 *mode* 是 `"eval"` 時節點型🄵由*ast.parse()* 生成。

`body` 是單個節點，是運算式型🄵的其中之一。

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**Interactive**(*body*)

單個互動式輸入，和 tut-interac 中所述的相似。當 *mode* 是 "single" 時節點型🄵由*ast.parse()* 生成。

body 是*陳述式節點 (statement nodes)* 的*list*。

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.**FunctionType**(*argtypes*, *returns*)

函式的舊式型🄵🄵解的表示法，因🄵 3.5 之前的 Python 版本不支援 **PEP 484** 🄵釋。當 *mode* 是 "func_type" 時節點型🄵由*ast.parse()* 生成。

這種型🄵的🄵解看起來像這樣：

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

argtypes 是*運算式節點*的*list*。

returns 是單個*運算式節點*。

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'), indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

在 3.8 版被加入.

文本 (Literals)

class ast.**Constant**(*value*)

一個常數值。Constant 文本的 value 屬性包含它所代表的 Python 物件。表示的值可以是*str*、*bytes*、*int*、*float*、*complex* 和*bool* 的實例，以及常數*None* 和*Ellipsis*。

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**FormattedValue**(*value*, *conversion*, *format_spec*)

表示 f 字串 (f-string) 中的單個格式化欄位的節點。如果字串包含單個格式欄位🄵且🄵有其他🄵容，則可以隔離 (isolate) 該節點，否則它將出現在*JoinedStr* 中。

- value 🄵任何運算式節點（例如文字、變數或函式呼叫）。
- conversion 是一個整數：
  - -1: 無格式化

  - 97 (`ord('a')`)：`!` `a` *ASCII* 格式化
  - 114 (`ord('r')`)：`!r` *repr()* 格式化
  - 115 (`ord('s')`)：`!s` *string* 格式化
- `format_spec` 是一個*JoinedStr* 節點，表示值的格式，若未指定格式則🄵 `None`。`conversion` 和 `format_spec` 可以同時設定。

class `ast.`**JoinedStr**(*values*)

一個 f 字串，包含一系列*FormattedValue* 和*Constant* 節點。

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'), indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))])))
```

class `ast.`**TemplateStr**(*values*, */*)

在 3.14 版被加入.

Node representing a template string literal, comprising a series of *Interpolation* and *Constant* nodes. These nodes may be any order, and do not need to be interleaved.

```
>>> expr = ast.parse('t"{name} finished {place:ordinal}"', mode='eval')
>>> print(ast.dump(expr, indent=4))
Expression(
    body=TemplateStr(
        values=[
            Interpolation(
                value=Name(id='name', ctx=Load()),
                str='name',
                conversion=-1),
            Constant(value=' finished '),
            Interpolation(
                value=Name(id='place', ctx=Load()),
                str='place',
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='ordinal')]))]))
```

class `ast.`**Interpolation**(*value*, *str*, *conversion*, *format_spec=None*)

在 3.14 版被加入.

Node representing a single interpolation field in a template string literal.

- `value` 🄵任何運算式節點（例如文字、變數或函式呼叫）。這和 `FormattedValue.value` 的意思相同。
- `str` is a constant containing the text of the interpolation expression.

- `conversion` 是一個整數：
  - -1：無規範
  - 97 (`ord('a')`)：`!a` *ASCII* 轉🄵
  - 114 (`ord('r')`)：`!r` *repr()* 轉🄵
  - 115 (`ord('s')`)：`!s` *string* 轉🄵

  這與 `FormattedValue.conversion` 的意思相同。
- `format_spec` 是一個 *JoinedStr* 節點，表示值的格式，若未指定格式則🄵 `None`。`conversion` 和 `format_spec` 可以同時設定。這與 `FormattedValue.format_spec` 的意思相同。

class `ast.`**List**(*elts*, *ctx*)

class `ast.`**Tuple**(*elts*, *ctx*)

串列或元組。`elts` 保存表示元素的節點串列。如果容器是賦值目標（即 `(x,y)=something`），則 `ctx` 是 *Store*，否則是 *Load*。

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class `ast.`**Set**(*elts*)

一個集合。`elts` 保存表示集合之元素的節點串列。

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class `ast.`**Dict**(*keys*, *values*)

一個字典 (dictionary)。`keys` 和 `values` 分🄵按匹配順序保存表示鍵和值的節點串列（🄵呼叫 `dictionary.keys()` 和 `dictionary.values()` 時將回傳的🄵容）。

當使用字典文本進行字典解包 (unpack) 時，要擴充的運算式位於 `values` 串列中，在 `keys` 中的相應位置有一個 `None`。

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

變數

class ast.**Name**(*id*, *ctx*)

一個變數名稱。id 將名稱以字串形式保存，且 ctx 是以下型🄵之一。

class ast.**Load**

class ast.**Store**

class ast.**Del**

變數參照可用於載入變數的值、🄵其分配新值或🄵除它。變數參照被賦予情境 (context) 來區分這些情🄵。

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.**Starred**(*value*, *ctx*)

一個 *var 變數參照。value 保存變數，通常是一個*Name* 節點。在使用 *args 建置*Call* 節點時必須使用此型🄵。

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))])
```

運算式

class ast.**Expr**(*value*)

當運算式（例如函式呼叫）本身作🄵陳述式出現且未使用或儲存其回傳值時，它將被包裝在此容器中。value 保存此區段 (section) 中的一個其他節點：*Constant*、*Name*、*Lambda*、*Yield* 或*YieldFrom*

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
```

(繼續下一頁)

(繼續上一頁)

```
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))])
```

class ast.**UnaryOp**(*op*, *operand*)

一元運算 (unary operation)。`op` 是運算子，`operand` 是任何運算式節點。

class ast.**UAdd**

class ast.**USub**

class ast.**Not**

class ast.**Invert**

一元運算子標記。*Not* 是 `not` 關鍵字、*Invert* 是 `~` 運算子。

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

二元運算 (binary operation)（如加法或除法）。`op` 是運算子、`left` 和 `right` 是任意運算式節點。

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**

class ast.**Sub**

class ast.**Mult**

class ast.**Div**

class ast.**FloorDiv**

class ast.**Mod**

class ast.**Pow**

class ast.**LShift**

class ast.**RShift**

class ast.**BitOr**

class ast.**BitXor**

class ast.**BitAnd**

class ast.**MatMult**

二元運算子 token。

class ast.**BoolOp**(*op*, *values*)

布林運算'or' 或'and'。`op` 是*Or* 或*And*。`values` 是有所涉及的值。使用同一運算子的連續操作（例如 `a or b or c`）會被折🄵🄵具有多個值的一個節點。

這不包括 `not`，它是一個*UnaryOp*。

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**

class ast.**Or**

布林運算子 token。

class ast.**Compare**(*left*, *ops*, *comparators*)

兩個或多個值的比較。`left` 是比較中的第一個值、`ops` 是運算子串列、`comparators` 是要比較的第一個元素之後值的串列。

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**

class ast.**NotEq**

class ast.**Lt**

class ast.**LtE**

class ast.**Gt**

class ast.**GtE**

class ast.**Is**

class ast.**IsNot**

class ast.**In**

class ast.**NotIn**

比較運算子 token。

class ast.**Call**(*func*, *args*, *keywords*)

一個函式呼叫。`func` 是該函式，通常是一個*Name* 或*Attribute* 物件。而在引數中：

- `args` 保存按位置傳遞的引數串列。
- `keywords` 保存一個*keyword* 物件串列，表示透過關鍵字傳遞的引數。

`args` 和 `keywords` 引數是可選的，預設🄴空串列。

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.**keyword**(*arg*, *value*)

函式呼叫或類🄴定義的關鍵字引數。`arg` 是參數名稱的原始字串，`value` 是要傳入的節點。

class ast.**IfExp**(*test*, *body*, *orelse*)

像是 `a if b else c` 之類的運算式。每個欄位都保存一個節點，因此在以下範例中，所有三個都是*Name* 節點。

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.**Attribute**(*value*, *attr*, *ctx*)

屬性的存取，例如 `d.keys`。`value` 是一個節點，通常是一個*Name*。`attr` 是一個屬性名稱的字串，`ctx` 根據屬性的作用方式可能是*Load*、*Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.**NamedExpr**(*target*, *value*)

一個附名運算式 (named expression)。該 AST 節點由賦值運算式運算子（也稱🄵海象運算子）🄵生。相對於*Assign* 節點之第一個引數可🄵多個節點，在這種情🄵下 `target` 和 `value` 都必須是單個節點。

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

在 3.8 版被加入.

下標 (Subscripting)

class ast.**Subscript**(*value*, *slice*, *ctx*)

一個下標，例如 `l[1]`。`value` 是下標物件（通常是序列或對映）。`slice` 是索引、切片或鍵。它可以是一個*Tuple* 🄵包含一個*Slice*。根據下標執行的操作不同，`ctx` 可以是*Load*、*Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

常規切片（形式🄵 `lower:upper` 或 `lower:upper:step`）。只能直接或者或者作🄵*Tuple* 的元素出現在*Subscript* 的 *slice* 欄位🄵。

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
```

(繼續下一頁)

(繼續上一頁)

```
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

綜合運算式 (comprehensions)

class ast.**ListComp**(*elt*, *generators*)

class ast.**SetComp**(*elt*, *generators*)

class ast.**GeneratorExp**(*elt*, *generators*)

class ast.**DictComp**(*key*, *value*, *generators*)

串列和集合綜合運算、生成器運算式和字典綜合運算。`elt`（或 `key` 和 `value`）是單個節點，表示各個項目會被求值 (evaluate) 的部分。

`generators` 是一個*`comprehension`* 節點的串列。

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

綜合運算中的一個 `for` 子句。`target` 是用於每個元素的參照 - 通常是*Name* 或*Tuple* 節點。`iter` 是要🄴代的物件。`ifs` 是測試運算式的串列：每個 `for` 子句可以有多個 `ifs`。

`is_async` 表示綜合運算式是非同步的（使用 `async for` 而不是 `for`）。該值🄴整數（0 或 1）。

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode='eval'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

陳述式

class ast.**Assign**(*targets*, *value*, *type_comment*)

一個賦值。targets 是節點串列，value 是單個節點。

targets 中的多個節點表示🄵每個節點分配相同的值。解包是透過在 targets 中放置一個*Tuple* 或*List* 來表示的。

type_comment

type_comment 是一個可選字串，其中的🄵解🄵型🄵🄵釋。

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

帶有型🄵🄵釋的賦值。target 是單個節點，可以是*Name*、*Attribute* 或*Subscript*。annotation 是🄵釋，例如*Constant* 或*Name* 節點。value 是單個可選節點。

simple 總會是 0（表示一個「🄵雜」目標）或 1（表示一個「簡單」目標）。一個「簡單」目標僅包含一個*Name* 節點，且不出現在括號之間；所有其他目標都被視🄵是🄴雜的。只有簡單目標會出現在模組和類🄵的 __annotations__ 字典中。

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
```

(繼續下一頁)

(繼續上一頁)

```
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.**AugAssign**(*target*, *op*, *value*)

增加賦值 (augmented assignment)，例如 `a += 1`。在下面的範例中，`target` 是 `x` 的*Name* 節點（帶有*Store* 情境），`op` 是*Add*，`value` 是一個值🄵 1 的*Constant*。

與*Assign* 的目標不同，`target` 屬性不能屬於*Tuple* 或*List* 類🄵。

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class ast.**Raise**(*exc*, *cause*)

一個 `raise` 陳述式。`exc` 是要引發的例外物件，通常是*Call* 或*Name*，若是獨立的 `raise` 則🄵 `None`。`cause` 是 `raise x from y` 中的可選部分 `y`。

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class ast.**Assert**(*test*, *msg*)

一個斷言 (assertion)。`test` 保存條件，例如*Compare* 節點。`msg` 保存失敗訊息。

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class ast.**Delete**(*targets*)

代表一個 `del` 陳述式。`targets` 是節點串列，例如*Name*、*Attribute* 或*Subscript* 節點。

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
```

(繼續下一頁)

(繼續上一頁)

```
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class ast.**Pass**

一個 pass 陳述式。

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.**TypeAlias**(*name*, *type_params*, *value*)

透過 type 陳述式建立的型🄵🄵名 *(type alias)*。name 是🄵名的名稱、type_params 是型🄵參數 *(type parameter)* 的串列、value 是型🄵🄵名的值。

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

在 3.12 版被加入.

其他僅適用於函式或🄵圈🄵部的陳述式將在其他部分中描述。

引入 (imports)

class ast.**Import**(*names*)

一個 import 陳述式。names 是*alias* 節點的串列。

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

代表 from x import y。module 是'from' 名稱的原始字串，前面🄵有任何的點 (dot)，或者對於諸如 from . import foo 之類的陳述式則🄵 None。level 是一個整數，保存相對引入的級🄵（0 表示🄵對引入）。

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class ast.**alias**(*name*, *asname*)

這兩個參數都是名稱的原始字串。如果要使用常規名稱，asname 可以🄵 None。

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

流程控制

備🄵

諸如 `else` 之類的可選子句如果不存在，則將被儲存🄵空串列。

class ast.**If**(*test*, *body*, *orelse*)

一個 `if` 陳述式。`test` 保存單個節點，例如*Compare* 節點。`body` 和 `orelse` 各自保存一個節點串列。

`elif` 子句在 AST 中🄵有特殊表示，而是在前一個子句的 `orelse` 部分中作🄵額外的*If* 節點出現。

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

一個 `for` 🄵圈。`target` 保存🄵圈賦予的變數，🄵單個*Name*、*Tuple*、*List*、*Attribute* 或*Subscript* 節點。`iter` 保存要🄵圈跑過的項目，也🄵單個節點。`body` 和 `orelse` 包含要執行的節點串列。如果🄵圈正常完成，則執行 `orelse` 中的🄵容，而不是透過 `break` 陳述式執行。

type_comment

`type_comment` 是一個可選字串，其中的🄵解🄵型🄵🄵釋。

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
```

(繼續下一頁)

(繼續上一頁)

```
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**While**(*test*, *body*, *orelse*)

一個 `while` 🄵圈。`test` 保存條件，例如*`Compare`* 節點。

```
>>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**Break**

class ast.**Continue**

`break` 和 `continue` 陳述式。

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
```

(繼續下一頁)

(繼續上一頁)

```
                    orelse=[
                        Continue()])])])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

try 區塊。除 handlers 是*ExceptHandler* 節點的串列外，其他所有屬性都是要執行之節點的串列。

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**TryStar**(*body*, *handlers*, *orelse*, *finalbody*)

try 區塊，後面跟著 except* 子句。這些屬性與*Try* 相同，但是 handlers 中的*ExceptHandler* 節點被直譯 (interpret) 🄵 except* 區塊而不是 except。

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except* Exception:
...     ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
```

(繼續下一頁)

(繼續上一頁)

```
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.11 版被加入.

class ast.**ExceptHandler**(*type*, *name*, *body*)

單個 except 子句。type 是會被匹配的例外型🄵，通常是一個*Name* 節點（或者 None 表示會捕捉到所有例外的 except: 子句）。name 是用於保存例外的名稱之原始字串，如果子句🄵有 as foo，則🄵 None。body 是節點串列。

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.**With**(*items*, *body*, *type_comment*)

一個 with 區塊。items 是表示情境管理器的*withitem* 節點串列，body 是情境🄵的縮進區塊。

type_comment

type_comment 是一個可選字串，其中的🄵解🄵型🄵🄵釋。

class ast.**withitem**(*context_expr*, *optional_vars*)

with 區塊中的單個情境管理器。context_expr 是情境管理器，通常是一個*Call* 節點。Optional_vars 是 as foo 部分的*Name*、*Tuple* 或*List*，或者如果不使用則🄵 None。

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...     something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
```

(繼續下一頁)

(繼續上一頁)

```
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())])])])
```

模式匹配 (pattern matching)

class ast.**Match**(*subject*, *cases*)

一個 `match` 陳述式。`subject` 保存匹配的主題（與案例匹配的物件），`cases` 包含具有不同案例的*match_case* 節點的可🄴代物件。

在 3.10 版被加入.

class ast.**match_case**(*pattern*, *guard*, *body*)

`match` 陳述式中的單個案例模式。`pattern` 包含主題將與之匹配的匹配模式。請注意，🄵模式生成的*AST* 節點與🄴運算式生成的節點不同，即使它們共享相同的語法。

`guard` 屬性包含一個運算式，如果模式與主題匹配，則將對該運算式求值。

`body` 包含一個節點串列，如果模式匹配🄵且🄵防護運算式 (guard expression) 的求值 (evaluate) 結果🄵真，則會執行該節點串列。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load())),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchValue**(*value*)

以相等性進行比較的匹配文本或值的模式。`value` 是一個運算式節點。允許值節點受到匹配陳述式文件中所述的限制。如果匹配主題等於求出值，則此模式成功。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchSingleton**(*value*)

按識🄴性 (identity) 進行比較的匹配文本模式。`value` 是要與 `None`、`True` 或 `False` 進行比較的單例 (singleton)。如果匹配主題是給定的常數，則此模式成功。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchSequence**(*patterns*)

匹配序列模式。如果主題是一個序列，`patterns` 包含與主題元素匹配的模式。如果子模式之一是 `MatchStar` 節點，則匹配可變長度序列，否則匹配固定長度序列。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
```

(繼續下一頁)

(繼續上一頁)

```
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchStar**(*name*)

以可變長度匹配序列模式匹配序列的其余部分。如果 name 不是 None，則如果整體序列模式成功，則包含其余序列元素的串列將綁定到該名稱。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

匹配對映模式。keys 是運算式節點的序列。patterns 是相應的模式節點序列。rest 是一個可選名稱，可以指定它來捕獲剩余的對映元素。允許的鍵運算式受到匹配陳述式文件中所述的限制。

如果主題是對映，所有求值出的鍵運算式都存在於對映中，🄵且與每個鍵對應的值與相應的子模式匹配，則此模式成功。如果 rest 不是 None，則如果整體對映模式成功，則包含其余對映元素的字典將綁定到該名稱。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
```

(繼續下一頁)

(繼續上一頁)

```
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

匹配類🄵模式。`cls` 是一個給定要匹配的名義類🄵 (nominal class) 的運算式。`patterns` 是要與類🄵定義的模式匹配屬性序列進行匹配的模式節點序列。`kwd_attrs` 是要匹配的附加屬性序列（在類🄵模式中指定🄵關鍵字引數），`kwd_patterns` 是相應的模式（在類🄵模式中指定🄵關鍵字的值）。

如果主題是指定類🄴的實例，所有位置模式都與相應的類🄴定義屬性匹配，🄵且任何指定的關鍵字屬性與其相應模式匹配，則此模式成功。

注意：類🄵可以定義一個回傳 self 的特性 (property)，以便將模式節點與正在匹配的實例進行匹配。一些🄵建型🄵也以這種方式匹配，如同匹配陳述式文件中所述。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
```

(繼續下一頁)

(繼續上一頁)

```
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchAs**(*pattern*, *name*)

匹配的「as 模式 (as-pattern)」，🄵捕獲模式 (capture pattern) 或通配模式 (wildcard pattern)。pattern 包含主題將與之匹配的匹配模式。如果模式🄵 None，則該節點代表捕獲模式（即裸名 (bare name)）🄵且始終會成功。

name 屬性包含模式成功時將綁定的名稱。如果 name 🄵 None，則 pattern 也必須🄵 None，🄵且節點代表通配模式。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchAs(),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

class ast.**MatchOr**(*patterns*)

匹配的「or 模式 (or-pattern)」。or 模式依次將其每個子模式與主題進行匹配，直到成功🄵止，然後 or 模式就會被認🄵是成功的。如果🄵有一個子模式成功，則 or 模式將失敗。patterns 屬性包含將與主題進行匹配的匹配模式節點串列。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
```

(繼續下一頁)

(繼續上一頁)

```
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

在 3.10 版被加入.

Type annotations

class ast.**TypeIgnore**(*lineno*, *tag*)

A `# type: ignore` comment located at *lineno*. *tag* is the optional tag specified by the form `# type: ignore <tag>`.

```
>>> print(ast.dump(ast.parse('x = 1 # type: ignore', type_comments=True), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[
        TypeIgnore(lineno=1, tag='')])
>>> print(ast.dump(ast.parse('x: bool = 1 # type: ignore[assignment]', type_
↪comments=True), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='x', ctx=Store()),
            annotation=Name(id='bool', ctx=Load()),
            value=Constant(value=1),
            simple=1)],
    type_ignores=[
        TypeIgnore(lineno=1, tag='[assignment]')])
```

備🄵

`TypeIgnore` nodes are not generated when the *type_comments* parameter is set to `False` (default). See *ast.parse()* for more details.

在 3.8 版被加入.

型🄵參數 (type parameters)

型🄵參數可以存在於類🄵、函式和型🄵🄵名上。

class ast.**TypeVar**(*name*, *bound*, *default_value*)

一個*typing.TypeVar*。`name` 是型🄵變數的名稱。`bound` 是（如果有存在的）界限 (bound) 或約束 (constraint)。如果 `bound` 是一個*Tuple*，它代表約束；否則它代表界限。`default_value` 🄵預設值；如果 `TypeVar` 🄵有預設值，那此屬性會被設🄵 `None`。

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] = list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool', ctx=Load()))],
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load()))])
```

在 3.12 版被加入.

在 3.13 版的變更: 新增 *default_value* 參數。

class ast.**ParamSpec**(*name*, *default_value*)

一個*typing.ParamSpec*。name 是參數規範的名稱。default_value 是預設值；如果 ParamSpec 🄵有預設值，則該屬性將設定🄵 None。

```
>>> print(ast.dump(ast.parse("type Alias[**P = [int, str]] = Callable[P, int]"),␣
↪indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=List(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
                        ctx=Load()))],
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

在 3.12 版被加入.

在 3.13 版的變更: 新增 *default_value* 參數。

class ast.**TypeVarTuple**(*name*, *default_value*)

一個*typing.TypeVarTuple*。name 是型🄵變數元組的名稱。default_value 🄵預設值；如果 TypeVarTuple 🄵有預設值，那此屬性會被設🄵 None。

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] = tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
```

(繼續下一頁)

(繼續上一頁)

```
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

在 3.12 版被加入.

在 3.13 版的變更: 新增 *default_value* 參數。

函式和類🄵定義

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

一個函式定義。

- `name` 是函式名稱的原始字串。
- `args` 是一個*arguments* 節點。
- `body` 是函式🄵節點的串列。
- `decorator_list` 是要應用的裝飾器串列，在最外層者會被儲存在首位（即串列中首位將會是最後一個被應用的那個）。
- `returns` 是回傳🄵釋。
- `type_params` 是型🄵參數的串列。

type_comment

`type_comment` 是一個可選字串，其中的🄵解🄵型🄵🄵釋。

在 3.12 版的變更: 新增了 `type_params`。

class ast.**Lambda**(*args*, *body*)

`lambda` 是可以在運算式🄵使用的最小函式定義。與*FunctionDef* 不同，`body` 保存單個節點。

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class ast.**arguments**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

函式的引數。

- `posonlyargs`、`args` 和 `kwonlyargs` 是*arg* 節點的串列。
- `vararg` 和 `kwarg` 是單個*arg* 節點，指的是 `*args`, `**kwargs` 參數。
- `kw_defaults` 是僅限關鍵字引數的預設值串列。如果其中某個🄵 `None`，則相應參數就會是必要的。
- `defaults` 是可以按位置傳遞的引數的預設值串列。如果預設值較少，則它們對應於最後 n 個引數。

class ast.**arg**(*arg*, *annotation*, *type_comment*)

串列中的單個引數。arg 是引數名稱的原始字串，annotation 是它的🄴釋，例如*Name* 節點。

type_comment

type_comment 是一個可選字串，其🄴解🄴型🄵🄵釋

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class ast.**Return**(*value*)

一個 return 陳述式。

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

一個 yield 或 yield from 運算式。因🄴這些是運算式，所以如果不使用發送回來的值，則必須將它們包裝在*Expr* 節點中。

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load())))])
```

(繼續下一頁)

(繼續上一頁)

```
>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load()))])])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

`global` 和 `nonlocal` 陳述式。`names` 是原始字串的串列。

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

一個類🄵定義。

- `name` 是類🄵名稱的原始字串
- `bases` 是被顯式指定的基底類🄵節點串列。
- `keywords` 是一個*keyword* 節點的串列，主要用於'metaclass'（元類🄵）。如 **PEP 3115** 所述，其他關鍵字將被傳遞到 metaclass。
- `body` 是表示類🄵定義中程式碼的節點串列。
- `decorator_list` 是一個節點串列，如*FunctionDef* 中所示。
- `type_params` 是型🄵參數的串列。

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
```

(繼續下一頁)

(繼續上一頁)

```
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())]))
```

在 3.12 版的變更: 新增了 `type_params`。

async 和 await

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

一個 `async def` 函式定義。與*FunctionDef* 具有相同的欄位。

在 3.12 版的變更: 新增了 `type_params`。

class ast.**Await**(*value*)

一個 `await` 運算式。`value` 是它等待的東西。僅在*AsyncFunctionDef* 主體 (body) 中有效。

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()))))])])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class ast.**AsyncWith**(*items*, *body*, *type_comment*)

`async for` 🄵圈和 `async with` 情境管理器。它們分🄵具有與*For* 和*With* 相同的欄位。僅在*AsyncFunctionDef* 主體中有效。

備🄵

當字串被*ast.parse()* 剖析時，回傳樹的運算子節點（`ast.operator`、`ast.unaryop`、`ast.cmpop`、`ast.boolop` 和 `ast.expr_context` 的子類🄵）將是單例。對其中之一的更改將反映在所有其他出現的相同值中（例如*ast.Add*）。

32.1.3 ast 輔助程式

除了節點類🄵之外，*ast* 模組還定義了這些用於遍歷 (traverse) 抽象語法樹的實用函式和類🄵:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*, *, *type_comments=False*, *feature_version=None*, *optimize=-1*)

將原始碼剖析🄵 AST 節點，相當於 `compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)`，其中 `FLAGS_VALUE` 在 `optimize <= 0` 時🄵 `ast.PyCF_ONLY_AST`，否則🄵 `ast.PyCF_OPTIMIZED_AST`。

如果給定 `type_comments=True`，剖析器將被修改🄵檢查🄵回傳 **PEP 484** 和 **PEP 526** 指定的型🄵🄵釋。這相當於將*ast.PyCF_TYPE_COMMENTS* 新增到傳遞給*compile()* 的旗標中。這將報告錯誤型🄵🄵釋的語法錯誤。如果🄵有此旗標，型🄵🄵釋將被忽略，🄵且所選 AST 節點上的 `type_comment`

欄位將始終🄵 None。此外，# type: ignore 🄵釋的位置將作🄵*Module* 的 type_ignores 屬性回傳（否則它始終是一個空串列）。

此外，如果 mode 是 'func_type'，則輸入語法將會依據 **PEP 484**「簽名型🄵🄵解 (signature type comments)」而被修改，例如 (str, int) -> List[str]。

將 feature_version 設定🄵元組 (major, minor) 將「盡可能」嘗試使用該 Python 版本的文法進行剖析。目前 major 必須等於 3。例如，設定 feature_version=(3, 9) 將嘗試禁止剖析 match 陳述式。目前 major 必須🄵 3、支援的最低版本🄵 (3, 7)（這在未來的 Python 版本中可能會增加）；最高的是 sys.version_info[0:2]。「盡可能」嘗試意味著不能保證剖析（或剖析的成功）與在與 feature_version 對應的 Python 版本上運行時相同。

如果來源包含 null 字元 (\0)，則會引發*ValueError*。

> **⚠ 警告**
>
> 請注意，成功將原始碼剖析🄵 AST 物件🄵不能保證提供的原始碼是可以執行的有效 Python 程式碼，因🄵編譯步驟可能會引發進一步的*SyntaxError* 例外。例如，原始的 return 42 🄵 return 陳述式生成一個有效的 AST 節點，但它不能單獨編譯（它需要位於函式節點🄵）。
>
> 特🄵是*ast.parse()* 不會執行任何範圍檢查，而編譯步驟才會執行此操作。

> **⚠ 警告**
>
> 由於 Python AST 編譯器中的堆🄵 (stack) 深度限制，太大或太🄵雜的字串可能會導致 Python 直譯器崩潰。

在 3.8 版的變更: 新增 type_comments、mode='func_type' 與 feature_version。

在 3.13 版的變更: feature_version 的最低支援版本現在是 (3, 7)。新增了 optimize 引數。

ast.**unparse**(*ast_obj*)

反剖析*ast.AST* 物件🄵生成一個帶有程式碼的字串，如果使用*ast.parse()* 剖析回來，該程式碼將生成等效的*ast.AST* 物件。

> **⚠ 警告**
>
> 生成的程式碼字串不一定等於生成*ast.AST* 物件的原始程式碼（🄵有任何編譯器最佳化，例如常數元組/凍結集合）。

> **⚠ 警告**
>
> 嘗試剖析高度🄵雜的運算式會導致*RecursionError*。

在 3.9 版被加入.

ast.**literal_eval**(*node_or_string*)

🄵僅包含 Python 文本或容器之顯示的運算式節點或字串來求值。提供的字串或節點只能包含以下 Python 文本結構：字串、位元組、數字、元組、串列、字典、集合、布林值、None 和 Ellipsis。

這可用於🄵包含 Python 值的字串求值，而無需自己剖析這些值。它無法計算任意🄵雜的運算式，例如涉及運算子或索引。

該函式過去被記🄵🄵「安全」，但🄵有定義其含義，這有點誤導讀者，它是特🄵設計🄵不去執行 Python 程式碼，與更通用的*eval()* 不同。🄵有命名空間、🄵有名稱查找、也🄵有呼叫的能力。但它

也不能免受攻擊：相對較小的輸入可能會導致記憶體耗盡或 C 堆🄵耗盡，從而導致行程崩潰。某些輸入也可能會出現 CPU 消耗過多而導致拒🄵服務的情🄵。因此不建議在不受信任的資料上呼叫它。

> **警告**
>
> 由於 Python AST 編譯器的堆🄵深度限制，Python 直譯器可能會崩潰。
>
> 它可能會引發*ValueError*、*TypeError*、*SyntaxError*、*MemoryError* 和*RecursionError*，具體取🄵於格式錯誤的輸入。

在 3.2 版的變更: 現在允許位元組和集合文本 (set literal)。

在 3.9 版的變更: 現在支援使用 `'set()'` 建立空集合。

在 3.10 版的變更: 對於字串輸入，前導空格和定位字元 (tab) 現在已被去除。

`ast.`**`get_docstring`**(*node*, *clean=True*)

回傳給定 *node* 的文件字串 (docstring)（必須是*FunctionDef*、*AsyncFunctionDef*、*ClassDef* 或*Module* 節點）或如果它🄵有文件字串則🄵 `None`。如果 *clean* 🄵 true，則使用*inspect.cleandoc()* 清理文件字串的縮排。

在 3.5 版的變更: 目前已支援*AsyncFunctionDef*。

`ast.`**`get_source_segment`**(*source*, *node*, ***, *padded=False*)

取得生成 *node* 的 *source* 的原始碼片段。如果某些位置資訊（*lineno*、*end_lineno*、*col_offset* 或*end_col_offset*）遺漏，則回傳 `None`。

如果 *padded* 🄵 `True`，則多列陳述式的第一列將用空格填充 (padded) 以匹配其原始位置。

在 3.8 版被加入.

`ast.`**`fix_missing_locations`**(*node*)

當你使用*compile()* 編譯節點樹時，對於每個有支援*lineno* 和*col_offset* 屬性之節點，編譯器預期他們的這些屬性都要存在。填入生成的節點相當繁瑣，因此該輔助工具透過將這些屬性設定🄵父節點的值，在尚未設定的地方遞🄵地新增這些屬性。它從 *node* 開始遞🄵地作用。

`ast.`**`increment_lineno`**(*node*, *n=1*)

將樹中從 *node* 開始的每個節點的列號和結束列號增加 *n*。這對於「移動程式碼」到檔案中的不同位置很有用。

`ast.`**`copy_location`**(*new_node*, *old_node*)

如果可行，將原始位置（*lineno*、*col_offset*、*end_lineno* 和*end_col_offset*）從 *old_node* 🄵🄵到 *new_node*，🄵回傳 *new_node*。

`ast.`**`iter_fields`**(*node*)

🄵 *node* 上存在的 `node._fields` 中的每個欄位生成一個 `(fieldname, value)` 元組。

`ast.`**`iter_child_nodes`**(*node*)

生成 *node* 的所有直接子節點，即作🄵節點的所有欄位以及作🄵節點串列欄位的所有項目。

`ast.`**`walk`**(*node*)

遞🄵地生成樹中從 *node* 開始的所有後代節點（包括 *node* 本身），不按指定順序。如果你只想就地修改節點而不關心情境，這非常有用。

`class` `ast.`**`NodeVisitor`**

節點🄵覽者基底類🄵，它遍歷抽象語法樹🄵🄵找到的每個節點呼叫🄵覽者函式。該函式可能會回傳一個由*visit()* 方法轉發的值。

這個類🄵應該被子類🄵化，子類🄵新增🄵覽者方法。

`visit`(*node*)

🄵覽一個節點。預設實作呼叫名🄵 `self.visit_`*classname* 的方法，其中 *classname* 是節點類🄵的名稱，或者在該方法不存在時呼叫*generic_visit()*。

generic_visit(*node*)

該🄵覽者對該節點的所有子節點呼叫*visit()*。

請注意，除非🄵覽者呼叫*generic_visit()* 或🄵覽它們本身，否則不會🄵覽具有自定義🄵覽者方法的節點之子節點。

visit_Constant(*node*)

處理所有常數節點。

如果你想在遍歷期間將變更應用 (apply) 於節點，請不要使用*NodeVisitor*。🄵此，有個允許修改的特殊遍歷🄵覽者工具*NodeTransformer*。

在 3.8 版之後被🄵用: `visit_Num()`、`visit_Str()`、`visit_Bytes()`、`visit_NameConstant()` 和 `visit_Ellipsis()` 方法現已🄵用，🄵且不會在未來的 Python 版本中被呼叫。新增*visit_Constant()* 方法來處理所有常數節點。

class ast.**NodeTransformer**

一個*NodeVisitor* 子類🄵，它會遍歷抽象語法樹🄴允許修改節點。

NodeTransformer 將遍歷 AST 🄵使用🄵覽者方法的回傳值來替🄵或🄵除舊節點。如果🄵覽者方法的回傳值🄵 `None`，則該節點將從其位置中🄵除，否則將被替🄵🄵回傳值。回傳值可能是原始節點，在這種情🄵下不會發生替🄵。

下面是一個示範用的 transformer，它將查找所有出現名稱 (`foo`) 🄵改寫🄵 `data['foo']`：

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

請記住，如果你正在操作的節點有子節點，你必須自己轉🄵子節點或先呼叫該節點的*generic_visit()* 方法。

對於屬於陳述式總集 (collection) 一部分的節點（適用於所有陳述式節點），🄵覽者還可以回傳節點串列，而不僅僅是單個節點。

如果*NodeTransformer* 引進了新節點（不屬於原始樹的一部分），但🄵有給它們提供位置資訊（例如*lineno*），則應使用新的子樹呼叫*fix_missing_locations()* 以重新計算位置資訊：

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

你通常會像這樣使用 transformer：

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*, *, *indent=None*, *show_empty=False*)

回傳 *node* 中樹的格式化傾印 (formatted dump)，這主要用於除錯。如果 *annotate_fields* 🄵 true（🄵預設值），則回傳的字串將顯示欄位的名稱和值。如果 *annotate_fields* 🄵 false，則透過省略明確的欄位名稱，結果字串將更加縮🄵簡潔。預設情🄵下，不會傾印列號和行偏移量等屬性。如果需要，可以設定 *include_attributes* 🄵 true。

如果 *indent* 是非負整數或字串，那🄵樹將使用該縮排級🄵來做漂亮印出 (pretty-print)。縮排級🄵 0、負數或 `""` 只會插入🄵列符號 (newlines)。`None`（預設值）代表選擇單列表示。使用正整數縮排可以在每個級🄵縮排相同數量的空格。如果 *indent* 是一個字串（例如 `"\t"`），則該字串用於縮排每個級🄵。

如果 *show_empty* 🄵 false（預設值），則輸出中將省略可選的空串列。可選的 `None` 值總是會被省略。

在 3.9 版的變更: 新增 *indent* 選項。

在 3.13 版的變更: 新增 *show_empty* 選項。

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[]))))],
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

32.1.4 編譯器旗標

可以將以下旗標傳遞給*compile()* 以變更對程式的編譯效果:

ast.PyCF_ALLOW_TOP_LEVEL_AWAIT

🄵用對最高階 `await`、`async for`、`async with` 和非同步綜合運算的支援。

在 3.8 版被加入.

ast.PyCF_ONLY_AST

生成🄵回傳抽象語法樹，而不是回傳已編譯的程式碼物件。

ast.PyCF_OPTIMIZED_AST

回傳的 AST 會根據*compile()* 或*ast.parse()* 中的 *optimize* 引數進行最佳化。

在 3.13 版被加入.

ast.PyCF_TYPE_COMMENTS

🄵用對 **PEP 484** 和 **PEP 526** 樣式型🄵🄵釋的支援 (`# type: <type>`, `# type: ignore <stuff>`)。

在 3.8 版被加入.

ast.compare(*a*, *b*, */*, ***, *compare_attributes=False*)

Recursively compares two ASTs.

compare_attributes affects whether AST attributes are considered in the comparison. If *compare_attributes* is `False` (default), then attributes are ignored. Otherwise they must all be equal. This option is useful to check whether the ASTs are structurally equal but differ in whitespace or similar details. Attributes include line numbers and column offsets.

在 3.14 版被加入.

32.1.5 命令列用法

在 3.9 版被加入.

ast 模組可以作🄵🄵本從命令列執行，可以像這樣簡單地做到：

```
python -m ast [-m <mode>] [-a] [infile]
```

以下選項可被接受：

-h, --help
顯示幫助訊息🄴退出。

-m <mode>

--mode <mode>
指定必須編譯哪種類型的程式碼，像是*parse()* 中的 *mode* 引數。

--no-type-comments
不要剖析型🄵🄴解。

-a, --include-attributes
包括列號和行偏移量等屬性。

-i <indent>

--indent <indent>
AST 中節點的縮進（空格數）。

--feature-version <version>
Python version in the format 3.x (for example, 3.10). Defaults to the current version of the interpreter.

在 3.14 版被加入.

-O <level>

--optimize <level>
Optimization level for parser. Defaults to no optimization.

在 3.14 版被加入.

--show-empty
Show empty lists and fields that are `None`. Defaults to not showing empty objects.

在 3.14 版被加入.

如果指定了 `infile`，則其🄵容將被剖析🄵 AST 🄵傾印 (dump) 到 stdout。否則會從 stdin 讀取🄵容。

也參考

Green Tree Snakes 是一個外部文件資源，提供了有關使用 Python AST 的詳細資訊。

ASTTokens 使用生成它們的原始碼中的標記和文本的位置來🄵釋 Python AST。這對於進行原始碼轉🄵的工具很有幫助。

leoAst.py 透過在 token 和 ast 節點之間插入雙向鏈結，統一了 python 程式的基於 token 和基於剖析樹的視圖。

LibCST 將程式碼剖析🄵具體語法樹 (Concrete Syntax Tree)，看起來像 ast 樹🄴保留所有格式詳細資訊。它對於建置自動重構 (codemod) 應用程式和 linter 非常有用。

Parso 是一個 Python 剖析器，支援不同 Python 版本的錯誤復原和往返剖析。Parso 還能🄵列出 Python 檔案中的多個語法錯誤。

32.2 symtable --- 存取編譯器的符號表

原始碼： Lib/symtable.py

符號表 (symbol table) 是在生成位元組碼 (bytecode) 之前由編譯器從 AST 生成的。符號表負責計算程式碼中每個識🄵器 (identifier) 的作用域。*symtable* 提供了一個介面來檢查這些表。

32.2.1 🄵生符號表

symtable.**symtable**(*code*, *filename*, *compile_type*)

回傳 Python 原始 *code* 的頂層*SymbolTable*。*filename* 是包含程式碼之檔案之名稱。*compile_type* 類似於*compile()* 的 *mode* 引數。

32.2.2 檢查符號表

class symtable.**SymbolTableType**

表明*SymbolTable* 物件型🄵的列舉。

MODULE = "module"

用於模組的符號表。

FUNCTION = "function"

用於函式的符號表。

CLASS = "class"

用於類🄵的符號表。

以下成員代表不同風格的🄵釋作用域。

ANNOTATION = "annotation"

如果 `from __future__ import annotations` 處於活動狀態，則用於🄵解。

TYPE_ALIAS = "type alias"

用於 `type` 結構的符號表。

TYPE_PARAMETERS = "type parameters"

用於泛型函式 (generic functions) 或泛型類🄵 (generic classes) 的符號表。

TYPE_VARIABLE = "type variable"

用於形式意義上 (formal sense) 的🄵結、約束元組 (constraint tuple) 或單一型🄵變數的預設值的符號表，即 TypeVar、TypeVarTuple 或 ParamSpec 物件（後兩者不支援🄵結或約束元組）。

在 3.13 版被加入.

class symtable.**SymbolTable**

一個區塊 (block) 的命名空間表 (namespace table) 。建構函式 (constructor) 🄵不公開。

get_type()

回傳符號表的類型。可能的值🄵*SymbolTableType* 列舉的成員。

在 3.12 版的變更: 新增了 `'annotation'`、`'TypeVar bound'`、`'type alias'` 和 `'type parameter'` 作🄵可能的回傳值。

在 3.13 版的變更: 回傳值是*SymbolTableType* 列舉的成員。

回傳字串的確切值在未來可能會發生變化，因此建議使用*SymbolTableType* 成員而不是寫死 (hard-coded) 字串。

get_id()

回傳表的識🄵器。

get_name()

回傳表的名稱。如果表用於類🄵，則這是類🄵的名稱；如果表用於函式，則這是函式的名稱；如果表是全域的，則🄵 `'top'`（*get_type()* 會回傳 `'module'`）。對於型🄵參數作用域（用於泛型類🄵、函式和型🄵🄵名），它是底層類🄵、函式或型🄵🄵名的名稱。對於型🄵🄵名作用域，它是型🄵🄵名的名稱。對於*TypeVar* 綁定作用域，它會是 `TypeVar` 的名稱。

get_lineno()

回傳此表所代表的區塊中第一行的編號。

is_optimized()

如果可以最佳化該表中的區域變數，則回傳 `True`。

is_nested()

如果區塊是巢狀類🄵或函式，則回傳 `True`。

has_children()

如果區塊🄵有巢狀命名空間，則回傳 `True`。這些可以透過*get_children()* 獲得。

get_identifiers()

回傳包含表中符號之名稱的視圖物件 (view object)。請參🄵視圖物件的文件。

lookup(*name*)

在表中查找 *name* 🄵回傳一個*Symbol* 實例。

get_symbols()

回傳表中名稱的*Symbol* 實例串列。

get_children()

回傳巢狀符號表的串列。

class symtable.**Function**

一個函式或方法的命名空間。該類🄵繼承自*SymbolTable*。

get_parameters()

回傳一個包含此函式參數名稱的元組 (tuple)。

get_locals()

回傳一個包含此函式中區域變數 (locals) 名稱的元組。

get_globals()

回傳一個包含此函式中全域變數 (globals) 名稱的元組。

get_nonlocals()

回傳一個包含此函式中明確宣告的非區域變數 (nonlocals) 名稱的元組。

get_frees()

回傳一個包含此函式中的自由（閉包）變數 *(free (closure) variables)* 名稱的元組。

class symtable.**Class**

一個類🄵的命名空間。該類🄵繼承自*SymbolTable*。

get_methods()

回傳一個包含類🄵中聲明的類似方法之函式名稱的元組。

在這🄵，術語「方法 (method)」表示透過 `def` 或 `async def` 在類🄵主體中定義的任何函式。

get_methods() 不會取得更深作用域🄵定義的函式（例如在🄵部類🄵中）。

舉例來🄵：

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...     def f():
...         def w(): pass
...
...     def g(self): pass
...
...     @classmethod
...     async def h(cls): pass
...
...     global outer
...     def outer(self): pass
... ''', 'test', 'exec')
>>> class_A = st.get_children()[2]
>>> class_A.get_methods()
('f', 'g', 'h')
```

🄵管 `A().f()` 會在 runtime 引發 *TypeError*，但 `A.f` 仍然被視🄵類似方法的函式。

Deprecated since version 3.14, will be removed in version 3.16.

class symtable.**Symbol**

SymbolTable 中的條目對應於來源中的識🄵器。建構函式不是公開的。

get_name()

回傳符號的名稱。

is_referenced()

如果該符號在其區塊中使用，則回傳 `True`。

is_imported()

如果符號是從 import 陳述式建立的，則回傳 `True`。

is_parameter()

如果符號是一個參數，則回傳 `True`。

is_type_parameter()

如果符號是一個型🄵參數，則回傳 `True`。

在 3.14 版被加入.

is_global()

如果符號是全域的，則回傳 `True`。

is_nonlocal()

如果符號是非區域的，則回傳 `True`。

is_declared_global()

如果使用全域陳述式將符號聲明🄵全域的，則回傳 `True`。

is_local()

如果符號是其區塊的區域符號，則回傳 `True`。

is_annotated()

如果符號有被🄵釋，則回傳 `True`。

在 3.6 版被加入.

is_free()

如果該符號在其區塊中被參照 (referenced) 但未被賦值 (assigned)，則回傳 `True`。

is_free_class()

Return *True* if a class-scoped symbol is free from the perspective of a method.

參考以下範例：

```
def f():
    x = 1  # function-scoped
    class C:
        x = 2  # class-scoped
        def method(self):
            return x
```

In this example, the class-scoped symbol `x` is considered to be free from the perspective of `C.method`, thereby allowing the latter to return *1* at runtime and not *2*.

在 3.14 版被加入.

is_assigned()

如果該符號被賦值到其區塊中，則回傳 `True`。

is_comp_iter()

Return `True` if the symbol is a comprehension iteration variable.

在 3.14 版被加入.

is_comp_cell()

Return `True` if the symbol is a cell in an inlined comprehension.

在 3.14 版被加入.

is_namespace()

如果名稱綁定引入 (introduce) 新的命名空間，則回傳 `True`。

如果名稱用作函式或類🄵陳述式的目標，則這將會是 true。

舉例來🄵：

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

請注意，單個名稱可以綁定到多個物件。如果結果🄵 `True`，則該名稱也可能被綁定到其他物件，例如 int 或 list，而不會引入新的命名空間。

get_namespaces()

回傳綁定到該名稱的命名空間的串列。

get_namespace()

回傳綁定到該名稱的命名空間。如果該名稱綁定了多個命名空間或🄵有命名空間，則會引發 *ValueError*。

32.2.3 命令列用法

在 3.13 版被加入.

symtable 模組可以從命令列作🄵🄵本執行。

```
python -m symtable [infile...]
```

🄵指定的 Python 原始檔🄵生符號表🄵轉儲 (dump) 到 stdout。如果未指定輸入檔案，則從 stdin 讀取🄵容。

32.3 `token` --- 與 Python 剖析樹一起使用的常數

原始碼： Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Tokens` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

Note that a token's value may depend on tokenizer options. For example, a `"+"` token may be reported as either *PLUS* or *OP*, or a `"match"` token may be either *NAME* or *SOFT_KEYWORD*.

`token.`**`tok_name`**

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

`token.`**`ISTERMINAL`**(*x*)

Return `True` for terminal token values.

`token.`**`ISNONTERMINAL`**(*x*)

Return `True` for non-terminal token values.

`token.`**`ISEOF`**(*x*)

Return `True` if *x* is the marker indicating the end of input.

The token constants are:

`token.`**`NAME`**

Token value that indicates an identifier. Note that keywords are also initially tokenized as `NAME` tokens.

`token.`**`NUMBER`**

Token value that indicates a numeric literal

`token.`**`STRING`**

Token value that indicates a string or byte literal, excluding formatted string literals. The token string is not interpreted: it includes the surrounding quotation marks and the prefix (if given); backslashes are included literally, without processing escape sequences.

`token.`**`OP`**

A generic token value that indicates an operator or delimiter.

CPython 實作細節： This value is only reported by the *tokenize* module. Internally, the tokenizer uses *exact token types* instead.

`token.`**`COMMENT`**

Token value used to indicate a comment. The parser ignores `COMMENT` tokens.

`token.`**`NEWLINE`**

Token value that indicates the end of a logical line.

`token.`**`NL`**

Token value used to indicate a non-terminating newline. `NL` tokens are generated when a logical line of code is continued over multiple physical lines. The parser ignores `NL` tokens.

`token.`**`INDENT`**

Token value used at the beginning of a logical line to indicate the start of an indented block.

`token.`**`DEDENT`**

Token value used at the beginning of a logical line to indicate the end of an indented block.

token.**FSTRING_START**

Token value used to indicate the beginning of an f-string literal.

CPython 實作細節： The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

token.**FSTRING_MIDDLE**

Token value used for literal text inside an f-string literal, including format specifications.

CPython 實作細節： Replacement fields (that is, the non-literal parts of f-strings) use the same tokens as other expressions, and are delimited by *LBRACE*, *RBRACE*, *EXCLAMATION* and *COLON* tokens.

token.**FSTRING_END**

Token value used to indicate the end of a f-string.

CPython 實作細節： The token string contains the closing quote(s).

token.**TSTRING_START**

Token value used to indicate the beginning of a template string literal.

CPython 實作細節： The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

在 3.14 版被加入.

token.**TSTRING_MIDDLE**

Token value used for literal text inside a template string literal including format specifications.

CPython 實作細節： Replacement fields (that is, the non-literal parts of t-strings) use the same tokens as other expressions, and are delimited by *LBRACE*, *RBRACE*, *EXCLAMATION* and *COLON* tokens.

在 3.14 版被加入.

token.**TSTRING_END**

Token value used to indicate the end of a template string literal.

CPython 實作細節： The token string contains the closing quote(s).

在 3.14 版被加入.

token.**ENDMARKER**

Token value that indicates the end of input. Used in top-level grammar rules.

token.**ENCODING**

Token value that indicates the encoding used to decode the source bytes into text. The first token returned by *tokenize.tokenize()* will always be an ENCODING token.

CPython 實作細節： This token type isn't used by the C tokenizer but is needed for the *tokenize* module.

The following token types are not produced by the *tokenize* module, and are defined for special uses in the tokenizer or parser:

token.**TYPE_IGNORE**

Token value indicating that a `type: ignore` comment was recognized. Such tokens are produced instead of regular *COMMENT* tokens only with the *PyCF_TYPE_COMMENTS* flag.

token.**TYPE_COMMENT**

Token value indicating that a type comment was recognized. Such tokens are produced instead of regular *COMMENT* tokens only with the *PyCF_TYPE_COMMENTS* flag.

token.**SOFT_KEYWORD**

Token value indicating a soft keyword.

The tokenizer never produces this value. To check for a soft keyword, pass a *NAME* token's string to *keyword.issoftkeyword()*.

token.**ERRORTOKEN**

Token value used to indicate wrong input.

The *tokenize* module generally indicates errors by raising exceptions instead of emitting this token. It can also emit tokens such as *OP* or *NAME* with strings that are later rejected by the parser.

The remaining tokens represent specific operators and delimiters. (The *tokenize* module reports these as *OP*; see `exact_type` in the *tokenize* documentation for details.)

Token	Value
token.**LPAR**	"("
token.**RPAR**	")"
token.**LSQB**	"["
token.**RSQB**	"]"
token.**COLON**	":"
token.**COMMA**	","
token.**SEMI**	";"
token.**PLUS**	"+"
token.**MINUS**	"-"
token.**STAR**	"*"
token.**SLASH**	"/"
token.**VBAR**	"|"
token.**AMPER**	"&"
token.**LESS**	"<"
token.**GREATER**	">"
token.**EQUAL**	"="

繼續下一頁

表格 1 – 繼續上一頁

Token	Value
token.**DOT**	"."
token.**PERCENT**	"%"
token.**LBRACE**	"{"
token.**RBRACE**	"}"
token.**EQEQUAL**	"=="
token.**NOTEQUAL**	"!="
token.**LESSEQUAL**	"<="
token.**GREATEREQUAL**	">="
token.**TILDE**	"~"
token.**CIRCUMFLEX**	"^"
token.**LEFTSHIFT**	"<<"
token.**RIGHTSHIFT**	">>"
token.**DOUBLESTAR**	"**"
token.**PLUSEQUAL**	"+="
token.**MINEQUAL**	"-="
token.**STAREQUAL**	"*="
token.**SLASHEQUAL**	"/="
token.**PERCENTEQUAL**	"%="

繼續下一頁

表格 1 – 繼續上一頁

Token	Value
token.**AMPEREQUAL**	"&="
token.**VBAREQUAL**	"|="
token.**CIRCUMFLEXEQUAL**	"^="
token.**LEFTSHIFTEQUAL**	"<<="
token.**RIGHTSHIFTEQUAL**	">>="
token.**DOUBLESTAREQUAL**	"**="
token.**DOUBLESLASH**	"//"
token.**DOUBLESLASHEQUAL**	"//="
token.**AT**	"@"
token.**ATEQUAL**	"@="
token.**RARROW**	"->"
token.**ELLIPSIS**	"..."
token.**COLONEQUAL**	":="
token.**EXCLAMATION**	"!"

The following non-token constants are provided:

token.**N_TOKENS**

The number of token types defined in this module.

token.**EXACT_TOKEN_TYPES**

A dictionary mapping the string representation of a token to its numeric code.

在 3.8 版被加入.

在 3.5 版的變更: Added AWAIT and ASYNC tokens.

在 3.7 版的變更: Added *COMMENT*, *NL* and *ENCODING* tokens.

在 3.7 版的變更: Removed AWAIT and ASYNC tokens. ”async” and ”await” are now tokenized as *NAME* tokens.

在 3.8 版的變更: Added *TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*. Added AWAIT and ASYNC tokens back (they're needed to support parsing older Python versions for *ast.parse()* with feature_version set to 6 or lower).

在 3.12 版的變更: 新增*EXCLAMATION*。

在 3.13 版的變更: Removed AWAIT and ASYNC tokens again.

32.4 keyword --- 檢驗 Python 關鍵字

原始碼： Lib/keyword.py

此模組允許 Python 程式確定某個字串是否🄵關鍵字或軟關鍵字 (soft keyword)。

keyword.**iskeyword**(*s*)

如果 *s* 是一個 Python 關鍵字則回傳 True。

keyword.**kwlist**

包含直譯器定義的所有 關鍵字的序列。如果所定義的任何關鍵字僅在特定*__future__* 陳述式生效時被🄵用，它們也將被包含在🄵。

keyword.**issoftkeyword**(*s*)

如果 *s* 是一個 Python 軟關鍵字則回傳 True。

在 3.9 版被加入.

keyword.**softkwlist**

包含直譯器定義的所有 軟關鍵字的序列。如果所定義的任何軟關鍵字僅在特定*__future__* 陳述式生效時被🄵用，它們也將被包含在🄵。

在 3.9 版被加入.

32.5 tokenize --- Tokenizer for Python source

原始碼： Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers", including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and *Ellipsis* are returned using the generic *OP* token type. The exact type can be determined by checking the exact_type property on the *named tuple* returned from *tokenize.tokenize()*.

⚠ 警告

Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using *ast.parse()*). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

32.5.1 Tokenizing Input

The primary entry point is a *generator*:

tokenize.**tokenize**(*readline*)

The *tokenize()* generator requires one argument, *readline*, which must be a callable object which provides the same interface as the *io.IOBase.readline()* method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *physical* line. The 5 tuple is returned as a *named tuple* with the field names: `type string start end line`.

The returned *named tuple* has an additional property named `exact_type` that contains the exact operator type for *OP* tokens. For all other token types `exact_type` equals the named tuple `type` field.

在 3.1 版的變更: 新增附名元組 (named tuple) 的支援。

在 3.3 版的變更: 新增 `exact_type` 的支援。

tokenize() determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

tokenize.**generate_tokens**(*readline*)

Tokenize a source reading unicode strings instead of bytes.

Like *tokenize()*, the *readline* argument is a callable returning a single line of input. However, *generate_tokens()* expects *readline* to return a str object rather than bytes.

The result is an iterator yielding named tuples, exactly like *tokenize()*. It does not yield an *ENCODING* token.

All constants from the *token* module are also exported from *tokenize*.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.**untokenize**(*iterable*)

Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the *ENCODING* token, which is the first token sequence output by *tokenize()*. If there is no encoding token in the input, it returns a str instead.

tokenize() needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

tokenize.**detect_encoding**(*readline*)

The *detect_encoding()* function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the *tokenize()* generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a *SyntaxError* will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use *open()* to open Python source files: it uses *detect_encoding()* to detect the file encoding.

tokenize.**open**(*filename*)

Open a file in read only mode using the encoding detected by *detect_encoding()*.

在 3.2 版被加入.

exception tokenize.**TokenError**

Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

或是:

```
[1,
 2,
 3
```

32.5.2 Command-Line Usage

在 3.3 版被加入.

The *tokenize* module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

可接受以下選項:

-h, --help

show this help message and exit

-e, --exact

display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

32.5.3 范例

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
```

(繼續下一頁)

(繼續上一頁)

```
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
```

(繼續下一頁)

(繼續上一頁)

```
4,0-4,0:            DEDENT          ''
4,0-4,9:            NAME            'say_hello'
4,9-4,10:           LPAR            '('
4,10-4,11:          RPAR            ')'
4,11-4,12:          NEWLINE         '\n'
5,0-5,0:            ENDMARKER       ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with *generate_tokens()*:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with *tokenize()*:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.6 tabnanny --- 偵測不良縮排

原始碼： Lib/tabnanny.py

目前現🄵是此模組打算以🄴本方式被呼叫使用，但也可以將其引入於 IDE 中🄵使用下方🄴述的*check()*函式。

> **備🄵**
>
> 此模組所提供的 API 很有可能會在未來的發🄵版本中有所變更，且有可能不具有向後相容性。

tabnanny.**check**(*file_or_dir*)

如果 *file_or_dir* 是個目🄵且🄵非符號鏈接 (symbolic link)，則會遞🄵地在名🄴 *file_or_dir* 的目🄵樹 (directory tree) 中不斷下行檢查所有 `.py` 檔案。如果 *file_or_dir* 是個一般 Python 原始檔案，則🄵其檢查空格相關問題。診斷訊息會以*print()* 函式輸出至標準輸出 (standard output) 當中。

tabnanny.**verbose**

標示是否要印出詳細訊息 (verbose message) 的旗標，若是以🄴本方式呼叫的話則可以用 `-v` 選項來增加。

tabnanny.**filename_only**

標示是否要只印出那些有空白相關問題檔案之檔名的旗標，若是以🄴本方式呼叫的話則可以用 `-q` 選項來設🄵真值。

exception tabnanny.**NannyNag**

當偵測到不良縮排時，此例外會被*process_tokens()* 引發，會在*check()* 中捕獲與處理。

tabnanny.**process_tokens**(*tokens*)

此函式被*check()* 用來處理由*tokenize* 🄵生的標記 (token)。

也參考

tokenize 模組
Python 原始程式碼的詞🄵掃描器 (lexical scanner)。

32.7 pyclbr --- Python 模組🄵覽器支援

原始碼： Lib/pyclbr.py

pyclbr 模組可以提供一個使用 Python 編寫的模組中定義的函式、類🄵和方法的有限資訊。這些資訊足以實作一個模組🄵覽器。這些資訊是從 Python 原始碼中抽取出來的，而不是透過引入模組，因此這個模組可以安全地與不信任的程式碼一起使用。此限制使得此模組無法與非 Python 實作的模組一起使用，包括所有標準與可選的擴充模組。

pyclbr.**readmodule**(*module*, *path=None*)

回傳模組層級的類🄵名稱與類🄵描述器對應的字典。如果可能的話，會包含引入基底類🄵的描述器。參數 *module* 是包含要讀取的模組名稱的字串；它可以是套件中的模組名稱。如果給定，*path* 是以 sys.path 🄵前綴的目🄵路徑序列，用來定位模組原始碼。

此函式🄵原始介面，僅保留用於向後相容性。它回傳的會是 readmodule_ex 回傳結果的篩選後版本。

pyclbr.**readmodule_ex**(*module*, *path=None*)

回傳一個以字典🄵基礎的樹狀結構，包含在模組中以 def 或 class 陳述式定義的每個函式和類🄵的函式或類🄵描述器。回傳的字典會將模組層級的函式和類🄵名稱對映到它們的描述器。巢狀物件會輸入其父物件的子字典。與 readmodule 一樣，*module* 會指定要讀取的模組，而 *path* 則會預先加入 sys.path。如果要讀取的模組是一個套件，回傳的字典有一個 key '__path__'，其值是一個包含套件搜尋路徑的串列。

在 3.7 版被加入: 巢狀定義（nested definitions）的描述器。這些描述器可透過新的子屬性來存取。每個描述器都有一個新的父屬性。

這些函式所回傳的描述器是 Function 和 Class 類🄵的實例。使用者不需要建立這些類🄵的實例。

32.7.1 函式物件

class pyclbr.**Function**

Class Function 實例描述由 def 陳述式定義的函式。它們具有以下屬性：

file
定義函式的檔案名稱。

module
定義所述函式的模組名稱。

name
函式的名稱。

lineno
檔案中定義開始的行號。

parent
對於頂層函式（top-level functions）🄵 None。對於巢狀函式，則使用父函式。

在 3.7 版被加入.

children
一個*dictionary* 將名稱對應到巢狀函式和類🄵的描述器。

在 3.7 版被加入.

is_async

對於使用 `async` 前綴定義的函式是 `True`，否則是 `False`。

在 3.10 版被加入.

32.7.2 類🄵物件

class pyclbr.**Class**

Class `Class` 實例描述由 class 陳述式定義的類🄵。它們具有與*Functions* 相同的屬性，以及另外兩個屬性。

file

定義類🄵的檔案名稱。

module

定義所述類🄵的模組名稱。

name

類🄵的名稱。

lineno

檔案中定義開始的行號。

parent

對於頂層類🄵🄵 `None`。對於巢狀類🄵，則使用父類🄵。

在 3.7 版被加入.

children

將名稱對應到巢狀函式和類🄵的描述器的字典。

在 3.7 版被加入.

super

描述被描述類🄵的直接基底類🄵的 `Class` 物件串列。被命名🄵超類🄵（superclasses）但無法被*readmodule_ex()* 發現的類🄵會以類🄵名稱的字串形式列出，而不是 `Class` 物件。

methods

一個*dictionary* 方法名稱對應到行號。這可以從較新的*children* 字典衍生出來，但🄵了向後相容性而保留。

32.8 py_compile —編譯 Python 來源檔案

原始碼： Lib/py_compile.py

The *py_compile* module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.**PyCompileError**

Exception raised when an error occurs while attempting to compile the file.

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147/PEP 488** path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used instead of *file* as the name of the

source file from which source lines are obtained for display in exception tracebacks. If *doraise* is true, a *PyCompileError* is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

The *doraise* and *quiet* arguments determine how errors are handled while compiling file. If *quiet* is 0 or 1, and *doraise* is false, the default behaviour is enabled: an error string is written to `sys.stderr`, and the function returns `None` instead of a path. If *doraise* is true, a *PyCompileError* is raised instead. However if *quiet* is 2, no message is written, and *doraise* has no effect.

If the path that *cfile* becomes (either explicitly specified or computed) is a symlink or non-regular file, *FileExistsError* will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

optimize controls the optimization level and is passed to the built-in *compile()* function. The default of `-1` selects the optimization level of the current interpreter.

invalidation_mode should be a member of the *PycInvalidationMode* enum and controls how the generated bytecode cache is invalidated at runtime. The default is *PycInvalidationMode.CHECKED_HASH* if the `SOURCE_DATE_EPOCH` environment variable is set, otherwise the default is *PycInvalidationMode.TIMESTAMP*.

在 3.2 版的變更: Changed default value of *cfile* to be **PEP 3147**-compliant. Previous default was *file* + `'c'` (`'o'` if optimization was enabled). Also added the *optimize* parameter.

在 3.4 版的變更: Changed code to use *importlib* for the byte-code cache file writing. This means file creation/writing semantics now match what *importlib* does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that *FileExistsError* is raised if *cfile* is a symlink or non-regular file.

在 3.7 版的變更: The *invalidation_mode* parameter was added as specified in **PEP 552**. If the `SOURCE_DATE_EPOCH` environment variable is set, *invalidation_mode* will be forced to *PycInvalidationMode.CHECKED_HASH*.

在 3.7.2 版的變更: The `SOURCE_DATE_EPOCH` environment variable no longer overrides the value of the *invalidation_mode* argument, and determines its default value instead.

在 3.8 版的變更: 新增 *quiet* 參數。

class py_compile.**PycInvalidationMode**

An enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See pyc-invalidation for more information on how Python invalidates `.pyc` files at runtime.

在 3.7 版被加入.

TIMESTAMP

The `.pyc` file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the `.pyc` file needs to be regenerated.

CHECKED_HASH

The `.pyc` file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the `.pyc` file needs to be regenerated.

UNCHECKED_HASH

Like *CHECKED_HASH*, the `.pyc` file includes a hash of the source file content. However, Python will at runtime assume the `.pyc` file is up to date and not validate the `.pyc` against the source file at all.

This option is useful when the `.pycs` are kept up to date by some system external to Python like a build system.

32.8.1 命令列介面

This module can be invoked as a script to compile several source files. The files named in *filenames* are compiled and the resulting bytecode is cached in the normal manner. This program does not search a directory structure to locate source files; it only compiles files named explicitly. The exit status is nonzero if one of the files could not be compiled.

<file> ... <fileN>

-

Positional arguments are files to compile. If - is the only parameter, the list of files is taken from standard input.

-q, --quiet

Suppress errors output.

在 3.2 版的變更: 新增對 - 的支援。

在 3.10 版的變更: 新增對-q 的支援。

也參考

***compileall* 模組**
Utilities to compile all Python source files in a directory tree.

32.9 compileall --- 位元組編譯 Python 函式庫

原始碼： Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

可用性: not WASI.

此模組在 WebAssembly 平台上不起作用或無法使用。更多資訊請參🄵*WebAssembly 平台*。

32.9.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...

file ...

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was -l *<directories from sys.path>*.

-l

Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f

Force rebuild even if timestamps are up-to-date.

-q

Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (-qq), all output is suppressed.

-d destdir

Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-s strip_prefix

Remove the given prefix from paths recorded in the `.pyc` files. Paths are made relative to the prefix.

This option can be used with `-p` but not with `-d`.

-p prepend_prefix

Prepend the given prefix to paths recorded in the `.pyc` files. Use `-p /` to make the paths absolute.

This option can be used with `-s` but not with `-d`.

-x regex

regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list

Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

-b

Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r

Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. **python -m compileall <directory> -r 0** is equivalent to **python -m compileall <directory> -l**.

-j N

Use *N* workers to compile the files within the given directory. If `0` is used, then the result of *os.process_cpu_count()* will be used.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]

Control how the generated byte-code files are invalidated at runtime. The `timestamp` value, means that `.pyc` files with the source timestamp and size embedded will be generated. The `checked-hash` and `unchecked-hash` values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the source file contents rather than a timestamp. See pyc-invalidation for more information on how Python validates bytecode cache files at runtime. The default is `timestamp` if the `SOURCE_DATE_EPOCH` environment variable is not set, and `checked-hash` if the `SOURCE_DATE_EPOCH` environment variable is set.

-o level

Compile with the given optimization level. May be used multiple times to compile for multiple levels at a time (for example, `compileall -o 1 -o 2`).

-e dir

Ignore symlinks pointing outside the given directory.

--hardlink-dupes

If two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

在 3.2 版的變更: 新增選項 `-i`、`-b` 與 `-h`。

在 3.5 版的變更: Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

在 3.7 版的變更: 新增選項 `--invalidation-mode`。

在 3.9 版的變更: Added the `-s`, `-p`, `-e` and `--hardlink-dupes` options. Raised the default recursion limit from 10 to *sys.getrecursionlimit()*. Added the possibility to specify the `-o` option multiple times.

There is no command-line option to control the optimization level used by the *compile()* function, because the Python interpreter itself already provides the option: **`python -O -m compileall`**.

Similarly, the *compile()* function respects the *sys.pycache_prefix* setting. The generated bytecode cache will only be useful if *compile()* is run with the same *sys.pycache_prefix* (if any) that will be used at runtime.

32.9.2 Public functions

`compileall.`**`compile_dir`**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `sys.getrecursionlimit()`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its `search` method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is 0, the number of cores in the system is used. If *workers* is lower than `0`, a *ValueError* will be raised.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

在 3.2 版的變更: 新增 *legacy* 與 *optimize* 參數。

在 3.5 版的變更: 新增 *workers* 參數。

在 3.5 版的變更: *quiet* parameter was changed to a multilevel value.

在 3.5 版的變更: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

在 3.6 版的變更: Accepts a *path-like object*.

在 3.7 版的變更: 新增 *invalidation_mode* 參數。

在 3.7.2 版的變更: 新增 *invalidation_mode* 參數的預設值被更新🄴 `None`。

在 3.8 版的變更: Setting *workers* to 0 now chooses the optimal number of cores.

在 3.9 版的變更: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments. Default value of *maxlevels* was changed from `10` to `sys.getrecursionlimit()`

`compileall.`**`compile_file`**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its `search` method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

在 3.2 版被加入.

在 3.5 版的變更: *quiet* parameter was changed to a multilevel value.

在 3.5 版的變更: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

在 3.7 版的變更: 新增 *invalidation_mode* 參數。

在 3.7.2 版的變更: 新增 *invalidation_mode* 參數的預設值被更新🄵 `None`。

在 3.9 版的變更: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments.

`compileall.`**`compile_path`**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

在 3.2 版的變更: 新增 *legacy* 與 *optimize* 參數。

在 3.5 版的變更: *quiet* parameter was changed to a multilevel value.

在 3.5 版的變更: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

在 3.7 版的變更: 新增 *invalidation_mode* 參數。

在 3.7.2 版的變更: 新增 *invalidation_mode* 參數的預設值被更新🄵 `None`。

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

也參考

py_compile **模組**
Byte-compile a single source file.

32.10 `dis` --- Python bytecode 的反組譯器

原始碼： Lib/dis.py

`dis` 模組支援反組譯分析 CPython *bytecode*。CPython bytecode 作🄵輸入的模組被定義於 `Include/opcode.h` 🄵且被編譯器和直譯器所使用。

CPython 實作細節： Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

在 3.6 版的變更: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

在 3.10 版的變更: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

在 3.11 版的變更: Some instructions are accompanied by one or more inline cache entries, which take the form of *`CACHE`* instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any *`dis`* utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

在 3.12 版的變更: The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction's *`CACHE`* entries.

As a consequence, the presence of the *`CACHE`* instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

在 3.13 版的變更: The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

在 3.14 版的變更: The *`-P`* command-line option and the `show_positions` argument were added.

The *`-S`* command-line option is added.

Example: Given the function `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST_BORROW         0 (alist)
              CALL                     1
              RETURN_VALUE
```

(The ”2” is a line number).

32.10.1 命令列介面

The *dis* module can be invoked as a script from the command line:

```
python -m dis [-h] [-C] [-O] [-P] [-S] [infile]
```

可接受以下選項:

-h, --help

Display usage and exit.

-C, --show-caches

Show inline caches.

在 3.13 版被加入.

-O, --show-offsets

Show offsets of instructions.

在 3.13 版被加入.

-P, --show-positions

Show positions of instructions in the source code.

在 3.14 版被加入.

-S, --specialized

Show specialized bytecode.

在 3.14 版被加入.

If `infile` is specified, its disassembled code will be written to stdout. Otherwise, disassembly is performed on compiled source code received from stdin.

32.10.2 Bytecode analysis

在 3.4 版被加入.

The bytecode analysis API allows pieces of Python code to be wrapped in a *Bytecode* object that provides easy access to details of the compiled code.

class dis.**Bytecode**(*x*, ***, *first_line=None*, *current_offset=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by *compile()*).

This is a convenience wrapper around many of the functions listed below, most notably *get_instructions()*, as iterating over a *Bytecode* instance yields the bytecode operations as *Instruction* instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means `dis()` will display a "current instruction" marker against the specified opcode.

If *show_caches* is `True`, `dis()` will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, `dis()` will display specialized bytecode that may be different from the original bytecode.

If *show_offsets* is `True`, `dis()` will include instruction offsets in the output.

If *show_positions* is `True`, `dis()` will include instruction source code positions in the output.

classmethod from_traceback(*tb*, ***, *show_caches=False*)

Construct a `Bytecode` instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by `dis.dis()`, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like `code_info()`.

在 3.7 版的變更: This can now handle coroutine and asynchronous generator objects.

在 3.11 版的變更: 新增 *show_caches* 與 *adaptive* 參數。

在 3.13 版的變更: 新增 *show_offsets* 參數。

在 3.14 版的變更: 新增 *show_positions* 參數。

範例:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
LOAD_GLOBAL
LOAD_FAST_BORROW
CALL
RETURN_VALUE
```

32.10.3 分析函式

The `dis` module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.**code_info**(*x*)

Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

在 3.2 版被加入.

在 3.7 版的變更: This can now handle coroutine and asynchronous generator objects.

dis.**show_code**(*x*, *, *file=None*)

Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

在 3.2 版被加入.

在 3.4 版的變更: 新增 *file* 參數。

dis.**dis**(*x=None*, *, *file=None*, *depth=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original bytecode.

在 3.4 版的變更: 新增 *file* 參數。

在 3.7 版的變更: Implemented recursive disassembling and added *depth* parameter.

在 3.7 版的變更: This can now handle coroutine and asynchronous generator objects.

在 3.11 版的變更: 新增 *show_caches* 與 *adaptive* 參數。

在 3.13 版的變更: 新增 *show_offsets* 參數。

在 3.14 版的變更: 新增 *show_positions* 參數。

dis.**distb**(*tb=None*, *, *file=None*, *show_caches=False*, *adaptive=False*, *show_offset=False*, *show_positions=False*)

Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

在 3.4 版的變更: 新增 *file* 參數。

在 3.11 版的變更: 新增 *show_caches* 與 *adaptive* 參數。

在 3.13 版的變更: 新增 *show_offsets* 參數。

在 3.14 版的變更: 新增 *show_positions* 參數。

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

dis.**disco**(*code*, *lasti=-1*, *, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the source code location of the instruction. Complete location information is shown if *show_positions* is true. Otherwise (the default) only the line number is displayed.

2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

在 3.4 版的變更: 新增 *file* 參數。

在 3.11 版的變更: 新增 *show_caches* 與 *adaptive* 參數。

在 3.13 版的變更: 新增 *show_offsets* 參數。

在 3.14 版的變更: 新增 *show_positions* 參數。

`dis.`**`get_instructions`**(*x*, ***, *first_line=None*, *show_caches=False*, *adaptive=False*)

Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of *`Instruction`* named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

The *adaptive* parameter works as it does in *`dis()`*.

在 3.4 版被加入.

在 3.11 版的變更: 新增 *show_caches* 與 *adaptive* 參數。

在 3.13 版的變更: The *show_caches* parameter is deprecated and has no effect. The iterator generates the *`Instruction`* instances with the *cache_info* field populated (regardless of the value of *show_caches*) and it no longer generates separate items for the cache entries.

`dis.`**`findlinestarts`**(*code*)

This generator function uses the `co_lines()` method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

在 3.6 版的變更: Line numbers can be decreasing. Before, they were always increasing.

在 3.10 版的變更: The **PEP 626** `co_lines()` method is used instead of the `co_firstlineno` and `co_lnotab` attributes of the code object.

在 3.13 版的變更: Line numbers can be `None` for bytecode that does not map to source lines.

`dis.`**`findlabels`**(*code*)

Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

`dis.`**`stack_effect`**(*opcode*, *oparg=None*, ***, *jump=None*)

Compute the stack effect of *opcode* with argument *oparg*.

If the code has a jump target and *jump* is `True`, *`stack_effect()`* will return the stack effect of jumping. If *jump* is `False`, it will return the stack effect of not jumping. And if *jump* is `None` (default), it will return the maximal stack effect of both cases.

在 3.4 版被加入.

在 3.8 版的變更: 新增 *jump* 參數。

在 3.13 版的變更: If `oparg` is omitted (or `None`), the stack effect is now returned for `oparg=0`. Previously this was an error for opcodes that use their arg. It is also no longer an error to pass an integer `oparg` when the `opcode` does not use it; the `oparg` in this case is ignored.

32.10.4 Python 位元組碼指令

The *`get_instructions()`* function and *`Bytecode`* class provide details of bytecode instructions as *`Instruction`* instances:

class dis.**Instruction**

位元組碼操作的詳細資訊

opcode

numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the *Opcode collections*.

opname

操作的可讀名稱

baseopcode

numeric code for the base operation if operation is specialized; otherwise equal to *`opcode`*

baseopname

human readable name for the base operation if operation is specialized; otherwise equal to *`opname`*

arg

numeric argument to operation (if any), otherwise `None`

oparg

`arg` 的🄵名。

argval

resolved arg value (if any), otherwise `None`

argrepr

human readable description of operation argument (if any), otherwise an empty string.

offset

start index of operation within bytecode sequence

start_offset

start index of operation within bytecode sequence, including prefixed `EXTENDED_ARG` operations if present; otherwise equal to *`offset`*

cache_offset

start index of the cache entries following the operation

end_offset

end index of the cache entries following the operation

starts_line

`True` if this opcode starts a source line, otherwise `False`

line_number

source line number associated with this opcode (if any), otherwise `None`

is_jump_target

`True` if other code jumps to here, otherwise `False`

jump_target

bytecode index of the jump target if this is a jump operation, otherwise `None`

positions

dis.Positions object holding the start and end locations that are covered by this instruction.

cache_info

Information about the cache entries of this instruction, as triplets of the form `(name, size, data)`, where the `name` and `size` describe the cache format and data is the contents of the cache. `cache_info` is `None` if the instruction does not have caches.

在 3.4 版被加入.

在 3.11 版的變更: Field `positions` is added.

在 3.13 版的變更: Changed field `starts_line`.

Added fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target`, `oparg`, `line_number` and `cache_info`.

class dis.**Positions**

In case the information is not available, some fields might be `None`.

lineno

end_lineno

col_offset

end_col_offset

在 3.11 版被加入.

The Python compiler currently generates the following bytecode instructions.

一般指令

In the following, We will refer to the interpreter stack as `STACK` and describe operations on it as if it was a Python list. The top of the stack corresponds to `STACK[-1]` in this language.

NOP

Do nothing code. Used as a placeholder by the bytecode optimizer, and to generate line tracing events.

NOT_TAKEN

Do nothing code. Used by the interpreter to record *BRANCH_LEFT* and *BRANCH_RIGHT* events for *sys.monitoring*.

在 3.14 版被加入.

POP_ITER

Removes the iterator from the top of the stack.

在 3.14 版被加入.

POP_TOP

Removes the top-of-stack item:

```
STACK.pop()
```

END_FOR

Removes the top-of-stack item. Equivalent to `POP_TOP`. Used to clean up at the end of loops, hence the name.

在 3.12 版被加入.

END_SEND

Implements `del STACK[-2]`. Used to clean up when a generator exits.

在 3.12 版被加入.

COPY (*i*)

Push the i-th item to the top of the stack without removing it from its original location:

```
assert i > 0
STACK.append(STACK[-i])
```

在 3.11 版被加入.

SWAP (*i*)

Swap the top of the stack with the i-th element:

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

在 3.11 版被加入.

CACHE

Rather than being an actual instruction, this opcode is used to mark extra space for the interpreter to cache useful data directly in the bytecode itself. It is automatically hidden by all `dis` utilities, but can be viewed with `show_caches=True`.

Logically, this space is part of the preceding instruction. Many opcodes expect to be followed by an exact number of caches, and will instruct the interpreter to skip over them at runtime.

Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

在 3.11 版被加入.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_NEGATIVE

實作 `STACK[-1] = -STACK[-1]`。

UNARY_NOT

實作 `STACK[-1] = not STACK[-1]`。

在 3.13 版的變更: This instruction now requires an exact *bool* operand.

UNARY_INVERT

實作 `STACK[-1] = ~STACK[-1]`。

GET_ITER

實作 `STACK[-1] = iter(STACK[-1])`。

GET_YIELD_FROM_ITER

If `STACK[-1]` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.

在 3.5 版被加入.

TO_BOOL

Implements `STACK[-1] = bool(STACK[-1])`.

在 3.13 版被加入.

Binary and in-place operations

Binary operations remove the top two items from the stack (`STACK[-1]` and `STACK[-2]`). They perform the operation, then put the result back on the stack.

In-place operations are like binary operations, but the operation is done in-place when `STACK[-2]` supports it, and the resulting `STACK[-1]` may be (but does not have to be) the original `STACK[-2]`.

BINARY_OP(*op*)

Implements the binary and in-place operators (depending on the value of *op*):

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

在 3.11 版被加入.

在 3.14 版的變更: With oparg :NB_SUBSCR, implements binary subscript (replaces opcode BINARY_SUBSCR)

STORE_SUBSCR

實作了:

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

實作了:

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

實作了:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
STACK.append(container[start:end])
```

在 3.12 版被加入.

STORE_SLICE

實作了:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
values = STACK.pop()
container[start:end] = value
```

在 3.12 版被加入.

Coroutine opcodes

GET_AWAITABLE(*where*)

Implements `STACK[-1] = get_awaitable(STACK[-1])`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the *CO_ITERABLE_COROUTINE* flag, or resolves `o.__await__`.

If the `where` operand is nonzero, it indicates where the instruction occurs:

- `1`: After a call to `__aenter__`
- `2`: After a call to `__aexit__`

在 3.5 版被加入.

在 3.11 版的變更: Previously, this instruction did not have an oparg.

GET_AITER

Implements `STACK[-1] = STACK[-1].__aiter__()`.

在 3.5 版被加入.

在 3.7 版的變更: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT

Implement `STACK.append(get_awaitable(STACK[-1].__anext__()))` to the stack. See `GET_AWAITABLE` for details about `get_awaitable`.

在 3.5 版被加入.

END_ASYNC_FOR

Terminates an `async for` loop. Handles an exception raised when awaiting a next item. The stack contains the async iterable in `STACK[-2]` and the raised exception in `STACK[-1]`. Both are popped. If the exception is not *StopAsyncIteration*, it is re-raised.

在 3.8 版被加入.

在 3.11 版的變更: Exception representation on the stack now consist of one, not three, items.

CLEANUP_THROW

Handles an exception raised during a `throw()` or `close()` call through the current frame. If `STACK[-1]` is an instance of *StopIteration*, pop three values from the stack and push its `value` member. Otherwise, re-raise `STACK[-1]`.

在 3.12 版被加入.

Miscellaneous opcodes

SET_ADD (*i*)

實作了:

```
item = STACK.pop()
set.add(STACK[-i], item)
```

Used to implement set comprehensions.

LIST_APPEND (*i*)

實作了:

```
item = STACK.pop()
list.append(STACK[-i], item)
```

Used to implement list comprehensions.

MAP_ADD (*i*)

實作了:

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

Used to implement dict comprehensions.

在 3.1 版被加入.

在 3.8 版的變更: Map value is `STACK[-1]` and map key is `STACK[-2]`. Before, those were reversed.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with `STACK[-1]` to the caller of the function.

YIELD_VALUE

Yields `STACK.pop()` from a *generator*.

在 3.11 版的變更: oparg set to be the stack depth.

在 3.12 版的變更: oparg set to be the exception block depth, for efficient closing of generators.

在 3.13 版的變更: oparg is `1` if this instruction is part of a yield-from or await, and `0` otherwise.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

在 3.6 版被加入.

POP_EXCEPT

Pops a value from the stack, which is used to restore the exception state.

在 3.11 版的變更: Exception representation on the stack now consist of one, not three, items.

RERAISE

Re-raises the exception currently on top of the stack. If oparg is non-zero, pops an additional value from the stack which is used to set `f_lasti` of the current frame.

在 3.9 版被加入.

在 3.11 版的變更: Exception representation on the stack now consist of one, not three, items.

PUSH_EXC_INFO

Pops a value from the stack. Pushes the current exception to the top of the stack. Pushes the value originally popped back to the stack. Used in exception handlers.

在 3.11 版被加入.

CHECK_EXC_MATCH

Performs exception matching for `except`. Tests whether the `STACK[-2]` is an exception matching `STACK[-1]`. Pops `STACK[-1]` and pushes the boolean result of the test.

在 3.11 版被加入.

CHECK_EG_MATCH

Performs exception matching for `except*`. Applies `split(STACK[-1])` on the exception group representing `STACK[-2]`.

In case of a match, pops two items from the stack and pushes the non-matching subgroup (`None` in case of full match) followed by the matching subgroup. When there is no match, pops one item (the match type) and pushes `None`.

在 3.11 版被加入.

WITH_EXCEPT_START

Calls the function in position 4 on the stack with arguments (type, val, tb) representing the exception at the top of the stack. Used to implement the call `context_manager.__exit__(*exc_info())` when an exception has occurred in a `with` statement.

在 3.9 版被加入.

在 3.11 版的變更: The `__exit__` function is in position 4 of the stack rather than 7. Exception representation on the stack now consist of one, not three, items.

LOAD_COMMON_CONSTANT

Pushes a common constant onto the stack. The interpreter contains a hardcoded list of constants supported by this instruction. Used by the `assert` statement to load *AssertionError*.

在 3.14 版被加入.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called to construct a class.

GET_LEN

Perform `STACK.append(len(STACK[-1]))`. Used in `match` statements where comparison with structure of pattern is needed.

在 3.10 版被加入.

MATCH_MAPPING

If `STACK[-1]` is an instance of *collections.abc.Mapping* (or, more technically: if it has the `Py_TPFLAGS_MAPPING` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

在 3.10 版被加入.

MATCH_SEQUENCE

If `STACK[-1]` is an instance of *collections.abc.Sequence* and is *not* an instance of *str*/*bytes*/*bytearray* (or, more technically: if it has the `Py_TPFLAGS_SEQUENCE` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

在 3.10 版被加入.

MATCH_KEYS

`STACK[-1]` is a tuple of mapping keys, and `STACK[-2]` is the match subject. If `STACK[-2]` contains all of the keys in `STACK[-1]`, push a *tuple* containing the corresponding values. Otherwise, push `None`.

在 3.10 版被加入.

在 3.11 版的變更: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

STORE_NAME(*namei*)

Implements `name = STACK.pop()`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use *STORE_FAST* or *STORE_GLOBAL* if possible.

DELETE_NAME(*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE(*count*)

Unpacks `STACK[-1]` into *count* individual values, which are put onto the stack right-to-left. Require there to be exactly *count* values.:

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX(*counts*)

Implements assignment with a starred target: Unpacks an iterable in `STACK[-1]` into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The number of values before and after the list value is limited to 255.

The number of values before the list value is encoded in the argument of the opcode. The number of values after the list if any is encoded using an `EXTENDED_ARG`. As a consequence, the argument can be seen as a two bytes values where the low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it.

The extracted values are put onto the stack right-to-left, i.e. `a, *b, c = d` will be stored after execution as `STACK.extend((a, b, c))`.

STORE_ATTR(*namei*)

實作了:

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

where *namei* is the index of name in `co_names` of the code object.

DELETE_ATTR(*namei*)

實作了：

```
obj = STACK.pop()
del obj.name
```

where *namei* is the index of name into `co_names` of the code object.

STORE_GLOBAL(*namei*)

Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL(*namei*)

Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST(*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_SMALL_INT(*i*)

Pushes the integer `i` onto the stack. `i` must be in `range(256)`

在 3.14 版被加入.

LOAD_NAME(*namei*)

Pushes the value associated with `co_names[namei]` onto the stack. The name is looked up within the locals, then the globals, then the builtins.

LOAD_LOCALS

Pushes a reference to the locals dictionary onto the stack. This is used to prepare namespace dictionaries for *LOAD_FROM_DICT_OR_DEREF* and *LOAD_FROM_DICT_OR_GLOBALS*.

在 3.12 版被加入.

LOAD_FROM_DICT_OR_GLOBALS(*i*)

Pops a mapping off the stack and looks up the value for `co_names[namei]`. If the name is not found there, looks it up in the globals and then the builtins, similar to *LOAD_GLOBAL*. This is used for loading global variables in annotation scopes within class bodies.

在 3.12 版被加入.

BUILD_TEMPLATE

Constructs a new *Template* instance from a tuple of strings and a tuple of interpolations and pushes the resulting object onto the stack:

```
interpolations = STACK.pop()
strings = STACK.pop()
STACK.append(_build_template(strings, interpolations))
```

在 3.14 版被加入.

BUILD_INTERPOLATION(*format*)

Constructs a new *Interpolation* instance from a value and its source expression and pushes the resulting object onto the stack.

If no conversion or format specification is present, `format` is set to `2`.

If the low bit of `format` is set, it indicates that the interpolation contains a format specification.

If `format >> 2` is non-zero, it indicates that the interpolation contains a conversion. The value of `format >> 2` is the conversion type (`0` for no conversion, `1` for `!s`, `2` for `!r`, and `3` for `!a`):

```
conversion = format >> 2
if format & 1:
    format_spec = STACK.pop()
else:
    format_spec = None
expression = STACK.pop()
value = STACK.pop()
STACK.append(_build_interpolation(value, expression, conversion, format_spec))
```

在 3.14 版被加入.

BUILD_TUPLE(*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack:

```
if count == 0:
    value = ()
else:
    value = tuple(STACK[-count:])
    STACK = STACK[:-count]

STACK.append(value)
```

BUILD_LIST(*count*)

Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET(*count*)

Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP(*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}`.

在 3.5 版的變更: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_STRING(*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

在 3.6 版被加入.

LIST_EXTEND(*i*)

實作了:

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

Used to build lists.

在 3.9 版被加入.

SET_UPDATE(*i*)

實作了:

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

Used to build sets.

在 3.9 版被加入.

DICT_UPDATE(*i*)

實作了:

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

Used to build dicts.

在 3.9 版被加入.

DICT_MERGE(*i*)

Like *DICT_UPDATE* but raises an exception for duplicate keys.

在 3.9 版被加入.

LOAD_ATTR(*namei*)

If the low bit of `namei` is not set, this replaces `STACK[-1]` with `getattr(STACK[-1], co_names[namei>>1])`.

If the low bit of `namei` is set, this will attempt to load a method named `co_names[namei>>1]` from the `STACK[-1]` object. `STACK[-1]` is popped. This bytecode distinguishes two cases: if `STACK[-1]` has a method with the correct name, the bytecode pushes the unbound method and `STACK[-1]`. `STACK[-1]` will be used as the first argument (`self`) by *CALL* or *CALL_KW* when calling the unbound method. Otherwise, `NULL` and the object returned by the attribute lookup are pushed.

在 3.12 版的變更: If the low bit of `namei` is set, then a `NULL` or `self` is pushed to the stack before the attribute or unbound method respectively.

LOAD_SUPER_ATTR(*namei*)

This opcode implements *super()*, both in its zero-argument and two-argument forms (e.g. `super().method()`, `super().attr` and `super(cls, self).method()`, `super(cls, self).attr`).

It pops three values from the stack (from top of stack down):

- `self`: the first argument to the current method
- `cls`: the class within which the current method was defined
- the global `super`

With respect to its argument, it works similarly to *LOAD_ATTR*, except that `namei` is shifted left by 2 bits instead of 1.

The low bit of `namei` signals to attempt a method load, as with *LOAD_ATTR*, which results in pushing `NULL` and the loaded method. When it is unset a single value is pushed to the stack.

The second-low bit of `namei`, if set, means that this was a two-argument call to *super()* (unset means zero-argument).

在 3.12 版被加入.

COMPARE_OP(*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname >> 5]`. If the fifth-lowest bit of `opname` is set (`opname & 16`), the result should be coerced to `bool`.

在 3.13 版的變更: The fifth-lowest bit of the oparg now indicates a forced conversion to *bool*.

IS_OP(*invert*)

Performs `is` comparison, or `is not` if `invert` is 1.

在 3.9 版被加入.

CONTAINS_OP(*invert*)

Performs `in` comparison, or `not in` if `invert` is 1.

在 3.9 版被加入.

IMPORT_NAME (*namei*)

Imports the module `co_names[namei]`. `STACK[-1]` and `STACK[-2]` are popped and provide the *fromlist* and *level* arguments of *__import__()*. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent *STORE_FAST* instruction modifies the namespace.

IMPORT_FROM (*namei*)

Loads the attribute `co_names[namei]` from the module found in `STACK[-1]`. The resulting object is pushed onto the stack, to be subsequently stored by a *STORE_FAST* instruction.

JUMP_FORWARD (*delta*)

Increments bytecode counter by *delta*.

JUMP_BACKWARD (*delta*)

Decrements bytecode counter by *delta*. Checks for interrupts.

在 3.11 版被加入.

JUMP_BACKWARD_NO_INTERRUPT (*delta*)

Decrements bytecode counter by *delta*. Does not check for interrupts.

在 3.11 版被加入.

POP_JUMP_IF_TRUE (*delta*)

If `STACK[-1]` is true, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

在 3.11 版的變更: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

在 3.12 版的變更: This is no longer a pseudo-instruction.

在 3.13 版的變更: This instruction now requires an exact *bool* operand.

POP_JUMP_IF_FALSE (*delta*)

If `STACK[-1]` is false, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

在 3.11 版的變更: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

在 3.12 版的變更: This is no longer a pseudo-instruction.

在 3.13 版的變更: This instruction now requires an exact *bool* operand.

POP_JUMP_IF_NOT_NONE (*delta*)

If `STACK[-1]` is not `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

在 3.11 版被加入.

在 3.12 版的變更: This is no longer a pseudo-instruction.

POP_JUMP_IF_NONE (*delta*)

If `STACK[-1]` is `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

在 3.11 版被加入.

在 3.12 版的變更: This is no longer a pseudo-instruction.

FOR_ITER (*delta*)

`STACK[-1]` is an *iterator*. Call its *__next__()* method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted then the byte code counter is incremented by *delta*.

在 3.12 版的變更: Up until 3.11 the iterator was popped when it was exhausted.

LOAD_GLOBAL (*namei*)

Loads the global named `co_names[namei>>1]` onto the stack.

在 3.11 版的變更: If the low bit of `namei` is set, then a `NULL` is pushed to the stack before the global variable.

LOAD_FAST(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

在 3.12 版的變更: This opcode is now only used in situations where the local variable is guaranteed to be initialized. It cannot raise *UnboundLocalError*.

LOAD_FAST_BORROW(*var_num*)

Pushes a borrowed reference to the local `co_varnames[var_num]` onto the stack.

在 3.14 版被加入.

LOAD_FAST_LOAD_FAST(*var_nums*)

Pushes references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

在 3.13 版被加入.

LOAD_FAST_BORROW_LOAD_FAST_BORROW(*var_nums*)

Pushes borrowed references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

在 3.14 版被加入.

LOAD_FAST_CHECK(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack, raising an *UnboundLocalError* if the local variable has not been initialized.

在 3.12 版被加入.

LOAD_FAST_AND_CLEAR(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack (or pushes `NULL` onto the stack if the local variable has not been initialized) and sets `co_varnames[var_num]` to `NULL`.

在 3.12 版被加入.

STORE_FAST(*var_num*)

Stores `STACK.pop()` into the local `co_varnames[var_num]`.

STORE_FAST_STORE_FAST(*var_nums*)

Stores `STACK[-1]` into `co_varnames[var_nums >> 4]` and `STACK[-2]` into `co_varnames[var_nums & 15]`.

在 3.13 版被加入.

STORE_FAST_LOAD_FAST(*var_nums*)

Stores `STACK.pop()` into the local `co_varnames[var_nums >> 4]` and pushes a reference to the local `co_varnames[var_nums & 15]` onto the stack.

在 3.13 版被加入.

DELETE_FAST(*var_num*)

Deletes local `co_varnames[var_num]`.

MAKE_CELL(*i*)

Creates a new cell in slot `i`. If that slot is nonempty then that value is stored into the new cell.

在 3.11 版被加入.

LOAD_DEREF(*i*)

Loads the cell contained in slot `i` of the "fast locals" storage. Pushes a reference to the object the cell contains on the stack.

在 3.11 版的變更: `i` is no longer offset by the length of `co_varnames`.

LOAD_FROM_DICT_OR_DEREF(*i*)

Pops a mapping off the stack and looks up the name associated with slot `i` of the "fast locals" storage in this mapping. If the name is not found there, loads it from the cell contained in slot `i`, similar to *LOAD_DEREF*. This is used for loading *closure variables* in class bodies (which previously used `LOAD_CLASSDEREF`) and in annotation scopes within class bodies.

在 3.12 版被加入.

STORE_DEREF(*i*)

Stores `STACK.pop()` into the cell contained in slot `i` of the "fast locals" storage.

在 3.11 版的變更: `i` is no longer offset by the length of `co_varnames`.

DELETE_DEREF(*i*)

Empties the cell contained in slot `i` of the "fast locals" storage. Used by the `del` statement.

在 3.2 版被加入.

在 3.11 版的變更: `i` is no longer offset by the length of `co_varnames`.

COPY_FREE_VARS(*n*)

Copies the `n` *free (closure) variables* from the closure into the frame. Removes the need for special code on the caller's side when calling closures.

在 3.11 版被加入.

RAISE_VARARGS(*argc*)

Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise STACK[-1]` (raise exception instance or type at `STACK[-1]`)
- 2: `raise STACK[-2] from STACK[-1]` (raise exception instance or type at `STACK[-2]` with `__cause__` set to `STACK[-1]`)

CALL(*argc*)

Calls a callable object with the number of arguments specified by `argc`. On the stack are (in ascending order):

- The callable
- `self` 或 `NULL`
- 剩余的位置引數

`argc` is the total of the positional arguments, excluding `self`.

`CALL` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

在 3.11 版被加入.

在 3.13 版的變更: The callable now always appears at the same position on the stack.

在 3.13 版的變更: Calls with keyword arguments are now handled by *CALL_KW*.

CALL_KW(*argc*)

Calls a callable object with the number of arguments specified by `argc`, including one or more named arguments. On the stack are (in ascending order):

- The callable
- `self` 或 `NULL`
- 剩余的位置引數
- The named arguments
- A *tuple* of keyword argument names

`argc` is the total of the positional and named arguments, excluding `self`. The length of the tuple of keyword argument names is the number of named arguments.

`CALL_KW` pops all arguments, the keyword names, and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

在 3.13 版被加入.

CALL_FUNCTION_EX(*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

在 3.6 版被加入.

PUSH_NULL

Pushes a `NULL` to the stack. Used in the call sequence to match the `NULL` pushed by `LOAD_METHOD` for non-method calls.

在 3.11 版被加入.

MAKE_FUNCTION

Pushes a new function object on the stack built from the code object at `STACK[-1]`.

在 3.10 版的變更: Flag value `0x04` is a tuple of strings instead of dictionary

在 3.11 版的變更: Qualified name at `STACK[-1]` was removed.

在 3.13 版的變更: Extra function attributes on the stack, signaled by oparg flags, were removed. They now use *SET_FUNCTION_ATTRIBUTE*.

SET_FUNCTION_ATTRIBUTE(*flag*)

Sets an attribute on a function object. Expects the function at `STACK[-1]` and the attribute value to set at `STACK[-2]`; consumes both and leaves the function at `STACK[-1]`. The flag determines which attribute to set:

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` a tuple of strings containing parameters' annotations
- `0x08` a tuple containing cells for free variables, making a closure

在 3.13 版被加入.

BUILD_SLICE(*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, implements:

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end))
```

if it is 3, implements:

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

See the *slice()* built-in function for more information.

EXTENDED_ARG(*ext*)

Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

CONVERT_VALUE(*oparg*)

Convert value to a string, depending on `oparg`:

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- `oparg == 1`: call *`str()`* on *value*
- `oparg == 2`: call *`repr()`* on *value*
- `oparg == 3`: call *`ascii()`* on *value*

Used for implementing formatted string literals (f-strings).

在 3.13 版被加入.

FORMAT_SIMPLE

Formats the value on top of stack:

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

在 3.13 版被加入.

FORMAT_WITH_SPEC

Formats the given value with the given format spec:

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

在 3.13 版被加入.

MATCH_CLASS(*count*)

`STACK[-1]` is a tuple of keyword attribute names, `STACK[-2]` is the class being matched against, and `STACK[-3]` is the match subject. *count* is the number of positional sub-patterns.

Pop `STACK[-1]`, `STACK[-2]`, and `STACK[-3]`. If `STACK[-3]` is an instance of `STACK[-2]` and has the positional and keyword attributes required by *count* and `STACK[-1]`, push a tuple of extracted attributes. Otherwise, push `None`.

在 3.10 版被加入.

在 3.11 版的變更: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

RESUME(*context*)

A no-op. Performs internal tracing, debugging and optimization checks.

The `context` operand consists of two parts. The lowest two bits indicate where the `RESUME` occurs:

- `0` The start of a function, which is neither a generator, coroutine nor an async generator
- `1` After a `yield` expression

- `2` After a `yield from` expression
- `3` After an `await` expression

The next bit is `1` if the RESUME is at except-depth `1`, and `0` otherwise.

在 3.11 版被加入.

在 3.13 版的變更: The oparg value changed to include information about except-depth

RETURN_GENERATOR

Create a generator, coroutine, or async generator from the current frame. Used as first opcode of in code object for the above mentioned callables. Clear the current frame and return the newly created generator.

在 3.11 版被加入.

SEND (*delta*)

Equivalent to `STACK[-1] = STACK[-2].send(STACK[-1])`. Used in `yield from` and `await` statements.

If the call raises *StopIteration*, pop the top value from the stack, push the exception's `value` attribute, and increment the bytecode counter by *delta*.

在 3.11 版被加入.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes in the range [0,255] which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

If your application uses pseudo instructions or specialized instructions, use the *hasarg* collection instead.

在 3.6 版的變更: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

在 3.12 版的變更: Pseudo instructions were added to the *dis* module, and for them it is not true that comparison with `HAVE_ARGUMENT` indicates whether they use their arg.

在 3.13 版之後被🄴用: Use *hasarg* instead.

CALL_INTRINSIC_1

Calls an intrinsic function with one argument. Passes `STACK[-1]` as the argument and sets `STACK[-1]` to the result. Used to implement functionality that is not performance critical.

The operand determines which intrinsic function is called:

運算元	描述
`INTRINSIC_1_INVAL`	Not valid
`INTRINSIC_PRINT`	Prints the argument to standard out. Used in the REPL.
`INTRINSIC_IMPORT_`	Performs `import *` for the named module.
`INTRINSIC_STOPITE`	Extracts the return value from a `StopIteration` exception.
`INTRINSIC_ASYNC_G`	Wraps an async generator value
`INTRINSIC_UNARY_P`	Performs the unary + operation
`INTRINSIC_LIST_TO_`	Converts a list to a tuple
`INTRINSIC_TYPEVAR`	Creates a *typing.TypeVar*
`INTRINSIC_PARAMSP`	Creates a *typing.ParamSpec*
`INTRINSIC_TYPEVAR`	Creates a *typing.TypeVarTuple*
`INTRINSIC_SUBSCRI`	Returns *typing.Generic* subscripted with the argument
`INTRINSIC_TYPEALI`	Creates a *typing.TypeAliasType*; used in the `type` statement. The argument is a tuple of the type alias's name, type parameters, and value.

在 3.12 版被加入.

CALL_INTRINSIC_2

Calls an intrinsic function with two arguments. Used to implement functionality that is not performance critical:

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

The operand determines which intrinsic function is called:

運算元	描述
INTRINSIC_2_INVALID	Not valid
INTRINSIC_PREP_RERAISE_STAR	Calculates the *ExceptionGroup* to raise from a try-except*.
INTRINSIC_TYPEVAR_WITH_BOUND	Creates a *typing.TypeVar* with a bound.
INTRINSIC_TYPEVAR_WITH_CONSTRAIN'	Creates a *typing.TypeVar* with constraints.
INTRINSIC_SET_FUNCTION_TYPE_PARAI	Sets the __type_params__ attribute of a function.

在 3.12 版被加入.

LOAD_SPECIAL

Performs special method lookup on STACK[-1]. If type(STACK[-1]).__xxx__ is a method, leave type(STACK[-1]).__xxx__; STACK[-1] on the stack. If type(STACK[-1]).__xxx__ is not a method, leave STACK[-1].__xxx__; NULL on the stack.

在 3.14 版被加入.

Pseudo-instructions

These opcodes do not appear in Python bytecode. They are used by the compiler but are replaced by real opcodes or removed before bytecode is generated.

SETUP_FINALLY (*target*)

Set up an exception handler for the following code block. If an exception occurs, the value stack level is restored to its current state and control is transferred to the exception handler at target.

SETUP_CLEANUP (*target*)

Like SETUP_FINALLY, but in case of an exception also pushes the last instruction (lasti) to the stack so that RERAISE can restore it. If an exception occurs, the value stack level and the last instruction on the frame are restored to their current state, and control is transferred to the exception handler at target.

SETUP_WITH (*target*)

Like SETUP_CLEANUP, but in case of an exception one more item is popped from the stack before control is transferred to the exception handler at target.

This variant is used in with and async with constructs, which push the return value of the context manager's __enter__() or __aenter__() to the stack.

POP_BLOCK

Marks the end of the code block associated with the last SETUP_FINALLY, SETUP_CLEANUP or SETUP_WITH.

LOAD_CONST_IMMORTAL (*consti*)

Works as *LOAD_CONST*, but is more efficient for immortal objects.

JUMP

JUMP_NO_INTERRUPT

Undirected relative jump instructions which are replaced by their directed (forward/backward) counterparts by the assembler.

JUMP_IF_TRUE

JUMP_IF_FALSE

Conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1`, `TO_BOOL`, `POP_JUMP_IF_TRUE/FALSE`.

LOAD_CLOSURE(*i*)

Pushes a reference to the cell contained in slot `i` of the "fast locals" storage.

Note that `LOAD_CLOSURE` is replaced with `LOAD_FAST` in the assembler.

在 3.13 版的變更: This opcode is now a pseudo-instruction.

32.10.5 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

在 3.12 版的變更: The collections now contain pseudo instructions and instrumented instructions as well. These are opcodes with values `>= MIN_PSEUDO_OPCODE` and `>= MIN_INSTRUMENTED_OPCODE`.

`dis.`**opname**

Sequence of operation names, indexable using the bytecode.

`dis.`**opmap**

Dictionary mapping operation names to bytecodes.

`dis.`**cmp_op**

Sequence of all compare operation names.

`dis.`**hasarg**

Sequence of bytecodes that use their argument.

在 3.12 版被加入.

`dis.`**hasconst**

Sequence of bytecodes that access a constant.

`dis.`**hasfree**

Sequence of bytecodes that access a *free (closure) variable*. 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes.

`dis.`**hasname**

Sequence of bytecodes that access an attribute by name.

`dis.`**hasjump**

Sequence of bytecodes that have a jump target. All jumps are relative.

在 3.13 版被加入.

`dis.`**haslocal**

Sequence of bytecodes that access a local variable.

`dis.`**hascompare**

Sequence of bytecodes of Boolean operations.

`dis.`**hasexc**

Sequence of bytecodes that set an exception handler.

在 3.12 版被加入.

`dis.`**hasjrel**

Sequence of bytecodes that have a relative jump target.

在 3.13 版之後被🄵用: All jumps are now relative. Use *hasjump*.

dis.**hasjabs**

Sequence of bytecodes that have an absolute jump target.

在 3.13 版之後被🄵用: All jumps are now relative. This list is empty.

32.11 pickletools --- pickle 開發者的工具

原始碼： Lib/pickletools.py

該模組包含與*pickle* 模組的詳細資訊相關的各種常數、一些有關實作的冗長🄵釋以及一些用於分析已 pickle 資料的有用函式。該模組的🄵容對於有*pickle* 相關工作的 Python 核心開發人員很有用；*pickle* 模組的一般使用者可能不會發現*pickletools* 模組。

32.11.1 命令列用法

在 3.2 版被加入.

當從命令列呼叫時，`python -m pickletools` 將拆解 (disassemble) 一個或多個 pickle 檔案的🄵容。請注意，如果你想查看儲存在 pickle 中的 Python 物件而不是 pickle 格式的詳細資訊，你可能需要使用 `-m pickle`。但是，當你要檢查的 pickle 檔案來自不受信任的來源時，`-m pickletools` 是一個更安全的選項，因🄵它不執行 pickle 位元組碼。

例如，pickle 於檔案 `x.pickle` 中的元組 `(1, 2)`：

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

命令列選項

-a, --annotate

用簡短的操作碼 (opcode) 描述🄵釋每一行。

-o, --output=<file>

應將輸出結果寫入之檔案的名稱。

-l, --indentlevel=<num>

新 MARK 級🄵縮進的空格數。

-m, --memo

當拆解多個物件時，會在拆解間保留備忘🄵。

-p, --preamble=<preamble>

當指定多個 pickle 檔案時，會在每次拆解之前印出給定的一段序言 (preamble)。

pickle_file

A pickle file to read, or - to indicate reading from standard input.

32.11.2 程式化介面

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

將 pickle 的符號拆解 (symbolic disassembly) 輸出到類檔案物件 *out*，預設🄵 `sys.stdout`。*pickle* 可以是字串或類檔案物件。*memo* 可以是一個 Python 字典，將用作 pickle 的備忘🄵；它可用於對同一 pickler 建立的多個 pickle 執行拆解。串流中由 `MARK` 操作碼指示的連續級🄵由 *indentlevel* 空格縮進。如果🄵 *annotate* 指定非零值，則輸出中的每個操作碼都會用簡短的描述進行🄵釋。*annotate* 的值用作🄵🄵釋之起始行提示。

在 3.2 版的變更: 新增 *annotate* 參數。

pickletools.**genops**(*pickle*)

提供 pickle 中所有操作碼的一個*iterator*，回傳形式🄵 `(opcode, arg, pos)` 三元組序列。*opcode* 是 `OpcodeInfo` 類🄵的實例；*arg* 是操作碼引數的解碼值（作🄵 Python 物件）；*pos* 是該操作碼所在的位置。*pickle* 可以是字串或類檔案物件。

pickletools.**optimize**(*picklestring*)

消除未使用的 `PUT` 操作碼後回傳一個新的等效 pickle 字串。最佳化後的 pickle 更短、傳輸時間更少、需要更小的儲存空間，且 unpickle 效率也更高。

CHAPTER 33

MS Windows 特有服務

此章節描述僅在 MS Windows 系統上可用的模組 (module)。

33.1 `msvcrt` --- MS VC++ runtime 提供的有用例程

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the *getpass* module uses this in the implementation of the *getpass()* function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

在 3.3 版的變更: Operations in this module now raise *OSError* where *IOError* was raised.

33.1.1 File Operations

`msvcrt.`**`locking`**(*fd*, *mode*, *nbytes*)

Lock part of a file based on file descriptor *fd* from the C runtime. Raises *OSError* on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

引發一個附帶引數 `fd`、`mode`、`nbytes` 的稽核事件 `msvcrt.locking`。

`msvcrt.`**`LK_LOCK`**

`msvcrt.`**`LK_RLCK`**

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, *OSError* is raised.

`msvcrt.`**`LK_NBLCK`**

`msvcrt.`**`LK_NBRLCK`**

Locks the specified bytes. If the bytes cannot be locked, *OSError* is raised.

msvcrt.**LK_UNLCK**

Unlocks the specified bytes, which must have been previously locked.

msvcrt.**setmode**(*fd*, *flags*)

Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be *os.O_TEXT*; for binary, it should be *os.O_BINARY*.

msvcrt.**open_osfhandle**(*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of *os.O_APPEND*, *os.O_RDONLY*, *os.O_TEXT* and *os.O_NOINHERIT*. The returned file descriptor may be used as a parameter to *os.fdopen()* to create a file object.

The file descriptor is inheritable by default. Pass *os.O_NOINHERIT* flag to make it non inheritable.

引發一個附帶引數 `handle`、`flags` 的稽核事件 `msvcrt.open_osfhandle`。

msvcrt.**get_osfhandle**(*fd*)

Return the file handle for the file descriptor *fd*. Raises *OSError* if *fd* is not recognized.

引發一個附帶引數 `fd` 的稽核事件 `msvcrt.get_osfhandle`。

33.1.2 Console I/O

msvcrt.**kbhit**()

Returns a nonzero value if a keypress is waiting to be read. Otherwise, return 0.

msvcrt.**getch**()

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.**getwch**()

Wide char variant of *getch()*, returning a Unicode value.

msvcrt.**getche**()

Similar to *getch()*, but the keypress will be echoed if it represents a printable character.

msvcrt.**getwche**()

Wide char variant of *getche()*, returning a Unicode value.

msvcrt.**putch**(*char*)

Print the byte string *char* to the console without buffering.

msvcrt.**putwch**(*unicode_char*)

Wide char variant of *putch()*, accepting a Unicode value.

msvcrt.**ungetch**(*char*)

Cause the byte string *char* to be ”pushed back” into the console buffer; it will be the next character read by *getch()* or *getche()*.

msvcrt.**ungetwch**(*unicode_char*)

Wide char variant of *ungetch()*, accepting a Unicode value.

33.1.3 Other Functions

msvcrt.**heapmin**()

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises *OSError*.

msvcrt.**set_error_mode**(*mode*)

Changes the location where the C runtime writes an error message for an error that might end the program. *mode* must be one of the `OUT_*` constants listed below or *REPORT_ERRMODE*. Returns the old setting or -1 if an error occurs. Only available in debug build of Python.

msvcrt.**OUT_TO_DEFAULT**

Error sink is determined by the app's type. Only available in debug build of Python.

msvcrt.**OUT_TO_STDERR**

Error sink is a standard error. Only available in debug build of Python.

msvcrt.**OUT_TO_MSGBOX**

Error sink is a message box. Only available in debug build of Python.

msvcrt.**REPORT_ERRMODE**

Report the current error mode value. Only available in debug build of Python.

msvcrt.**CrtSetReportMode**(*type*, *mode*)

Specifies the destination or destinations for a specific report type generated by `_CrtDbgReport()` in the MS VC++ runtime. *type* must be one of the `CRT_*` constants listed below. *mode* must be one of the `CRTDBG_*` constants listed below. Only available in debug build of Python.

msvcrt.**CrtSetReportFile**(*type*, *file*)

After you use *CrtSetReportMode()* to specify *CRTDBG_MODE_FILE*, you can specify the file handle to receive the message text. *type* must be one of the `CRT_*` constants listed below. *file* should be the file handle your want specified. Only available in debug build of Python.

msvcrt.**CRT_WARN**

Warnings, messages, and information that doesn't need immediate attention.

msvcrt.**CRT_ERROR**

Errors, unrecoverable problems, and issues that require immediate attention.

msvcrt.**CRT_ASSERT**

Assertion failures.

msvcrt.**CRTDBG_MODE_DEBUG**

Writes the message to the debugger's output window.

msvcrt.**CRTDBG_MODE_FILE**

Writes the message to a user-supplied file handle. *CrtSetReportFile()* should be called to define the specific file or stream to use as the destination.

msvcrt.**CRTDBG_MODE_WNDW**

Creates a message box to display the message along with the `Abort`, `Retry`, and `Ignore` buttons.

msvcrt.**CRTDBG_REPORT_MODE**

Returns current *mode* for the specified *type*.

msvcrt.**CRT_ASSEMBLY_VERSION**

The CRT Assembly version, from the `crtassem.h` header file.

msvcrt.**VC_ASSEMBLY_PUBLICKEYTOKEN**

The VC Assembly public key token, from the `crtassem.h` header file.

msvcrt.**LIBRARIES_ASSEMBLY_NAME_PREFIX**

The Libraries Assembly name prefix, from the `crtassem.h` header file.

33.2 winreg --- Windows 🄵🄵表存取

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

在 3.3 版的變更: Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

33.2.1 函式

This module offers the following functions:

`winreg.`**`CloseKey`**(*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> **備🄵**
>
> If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

`winreg.`**`ConnectRegistry`**(*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

引發一個附帶引數 `computer_name`、`key` 的稽核事件 `winreg.ConnectRegistry`。

在 3.3 版的變更: See *above*.

`winreg.`**`CreateKey`**(*key*, *sub_key*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

引發一個附帶引數 `key`、`sub_key`、`access` 的稽核事件 `winreg.CreateKey`。

引發一個附帶引數 `key` 的稽核事件 `winreg.OpenKey/result`。

在 3.3 版的變更: See *above*.

`winreg.`**`CreateKeyEx`**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

引發一個附帶引數 `key`、`sub_key`、`access` 的稽核事件 `winreg.CreateKey`。

引發一個附帶引數 `key` 的稽核事件 `winreg.OpenKey/result`。

在 3.2 版被加入.

在 3.3 版的變更: See *above*.

`winreg.`**`DeleteKey`**(*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

引發一個附帶引數 `key`、`sub_key`、`access` 的稽核事件 `winreg.DeleteKey`。

在 3.3 版的變更: See *above*.

`winreg.`**`DeleteKeyEx`**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WOW64_64KEY*. On 32-bit Windows, the WOW64 constants are ignored. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

On unsupported Windows versions, *NotImplementedError* is raised.

引發一個附帶引數 `key`、`sub_key`、`access` 的稽核事件 `winreg.DeleteKey`。

在 3.2 版被加入.

在 3.3 版的變更: See *above*.

`winreg.`**`DeleteValue`**(*key*, *value*)

Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

引發一個附帶引數 `key`、`value` 的稽核事件 `winreg.DeleteValue`。

winreg.**EnumKey**(*key*, *index*)

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an *OSError* exception is raised, indicating, no more values are available.

引發一個附帶引數 key、index 的稽核事件 winreg.EnumKey。

在 3.3 版的變更: See *above*.

winreg.**EnumValue**(*key*, *index*)

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an *OSError* exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	含義
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for *SetValueEx()*)

引發一個附帶引數 key、index 的稽核事件 winreg.EnumValue。

在 3.3 版的變更: See *above*.

winreg.**ExpandEnvironmentStrings**(*str*)

Expands environment variable placeholders %NAME% in strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

引發一個附帶引數 str 的稽核事件 winreg.ExpandEnvironmentStrings。

winreg.**FlushKey**(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call *FlushKey()* to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike *CloseKey()*, the *FlushKey()* method returns only when all the data has been written to the registry. An application should only call *FlushKey()* if it requires absolute certainty that registry changes are on disk.

> **備🄵**
>
> If you don't know whether a *FlushKey()* call is required, it probably isn't.

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by *ConnectRegistry()* or one of the constants *HKEY_USERS* or *HKEY_LOCAL_MACHINE*.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different from permissions -- see the RegLoadKey documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

引發一個附帶引數 `key`、`sub_key`、`file_name` 的稽核事件 `winreg.LoadKey`。

`winreg.`**`OpenKey`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

`winreg.`**`OpenKeyEx`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_READ`. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, `OSError` is raised.

引發一個附帶引數 `key`、`sub_key`、`access` 的稽核事件 `winreg.OpenKey`。

引發一個附帶引數 `key` 的稽核事件 `winreg.OpenKey/result`。

在 3.2 版的變更: Allow the use of named arguments.

在 3.3 版的變更: See *above*.

`winreg.`**`QueryInfoKey`**(*key*)

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items:

Index	含義
`0`	An integer giving the number of sub keys this key has.
`1`	An integer giving the number of values this key has.
`2`	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

引發一個附帶引數 `key` 的稽核事件 `winreg.QueryInfoKey`。

`winreg.`**`QueryValue`**(*key*, *sub_key*)

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the `SetValue()` method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a `NULL` name. But the underlying API call doesn't return the type, so always use `QueryValueEx()` if possible.

引發一個附帶引數 `key`、`sub_key`、`value_name` 的稽核事件 `winreg.QueryKey`。

`winreg.`**`QueryValueEx`**(*key*, *value_name*)

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	含義
`0`	The value of the registry item.
`1`	An integer giving the registry type for this value (see table in docs for *`SetValueEx()`*)

引發一個附帶引數 `key`、`sub_key`、`value_name` 的稽核事件 `winreg.QueryKey`。

`winreg.`**`SaveKey`**(*key*, *file_name*)

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the *`LoadKey()`* method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the **SeBackupPrivilege** security privilege. Note that privileges are different than permissions -- see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes `NULL` for *security_attributes* to the API.

引發一個附帶引數 `key`、`file_name` 的稽核事件 `winreg.SaveKey`。

`winreg.`**`SetValue`**(*key*, *sub_key*, *type*, *value*)

Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be *`REG_SZ`*, meaning only strings are supported. Use the *`SetValueEx()`* function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with *`KEY_SET_VALUE`* access.

引發一個附帶引數 `key`、`sub_key`、`type`、`value` 的稽核事件 `winreg.SetValue`。

`winreg.`**`SetValueEx`**(*key*, *value_name*, *reserved*, *type*, *value*)

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything -- zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with *KEY_SET_VALUE* access.

To open the key, use the *CreateKey()* or *OpenKey()* methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

引發一個附帶引數 `key`、`sub_key`、`type`、`value` 的稽核事件 `winreg.SetValue`。

`winreg.`**`DisableReflectionKey`**(*key*)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise *NotImplementedError* if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

引發一個附帶引數 `key` 的稽核事件 `winreg.DsiableReflectionKey`。

`winreg.`**`EnableReflectionKey`**(*key*)

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise *NotImplementedError* if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

引發一個附帶引數 `key` 的稽核事件 `winreg.EnableReflectionKey`。

`winreg.`**`QueryReflectionKey`**(*key*)

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise *NotImplementedError* if executed on a 32-bit operating system.

引發一個附帶引數 `key` 的稽核事件 `winreg.QueryReflectionKey`。

33.2.2 常數

The following constants are defined for use in many *winreg* functions.

HKEY_* Constants

`winreg.`**`HKEY_CLASSES_ROOT`**

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.`**`HKEY_CURRENT_USER`**

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.`**`HKEY_LOCAL_MACHINE`**

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.`**`HKEY_USERS`**

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

winreg.**HKEY_PERFORMANCE_DATA**

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

winreg.**HKEY_CURRENT_CONFIG**

Contains information about the current hardware profile of the local computer system.

winreg.**HKEY_DYN_DATA**

This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

winreg.**KEY_ALL_ACCESS**

Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

winreg.**KEY_WRITE**

Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

winreg.**KEY_READ**

Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

winreg.**KEY_EXECUTE**

Equivalent to *KEY_READ*.

winreg.**KEY_QUERY_VALUE**

Required to query the values of a registry key.

winreg.**KEY_SET_VALUE**

Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**

Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**

Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**

Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**

Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view. On 32-bit Windows, this constant is ignored.

winreg.**KEY_WOW64_32KEY**

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view. On 32-bit Windows, this constant is ignored.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**
: Binary data in any form.

winreg.**REG_DWORD**
: 32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**
: A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**
: A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**
: Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**
: A Unicode symbolic link.

winreg.**REG_MULTI_SZ**
: A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**
: No defined value type.

winreg.**REG_QWORD**
: A 64-bit number.

  在 3.6 版被加入.

winreg.**REG_QWORD_LITTLE_ENDIAN**
: A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

  在 3.6 版被加入.

winreg.**REG_RESOURCE_LIST**
: A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
: A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
: A hardware resource list.

winreg.**REG_SZ**
: A null-terminated string.

33.2.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` -- thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

引發一個附帶引數 `key` 的稽核事件 `winreg.PyHKEY.Detach`。

PyHKEY.**__enter__**()

PyHKEY.**__exit__**(**exc_info*)

The HKEY object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the `with` block.

33.3 `winsound` --- Windows 的音效播放介面

winsound 模組提供存取 Windows 平台所提供的基本音效播放機制。它包含數個函式與常數。

winsound.**Beep**(*frequency*, *duration*)

讓電腦的喇叭發出🄵聲。*frequency* 參數指定音效的頻率（以赫🄵🄵單位），範圍必須介於 37 到 32,767 之間。*duration* 參數指定音效持續的時間（以毫秒🄵單位）。若系統無法讓喇叭發出🄵聲，則會引發*RuntimeError*。

winsound.**PlaySound**(*sound*, *flags*)

呼叫平台 API 中底層的 `PlaySound()` 函式。*sound* 參數可以是檔案名稱、系統音效🄵名、作🄵*bytes-like object* 的音訊資料，或 `None`。其直譯方式取🄵於 *flags* 的值，該值可以是以下所述常數的位元 OR 組合。若 *sound* 🄵 `None`，則會停止任何正在播放的波形音效。若系統回報錯誤，則會引發*RuntimeError*。

winsound.**MessageBeep**(*type=MB_OK*)

呼叫平台 API 中底層的 `MessageBeep()` 函式。此函式會播放登🄵檔中指定的音效。*type* 引數指定要播放的音效類型，可接受的值包括 `-1`、`MB_ICONASTERISK`、`MB_ICONEXCLAMATION`、`MB_ICONHAND`、`MB_ICONQUESTION` 與 `MB_OK`，這些皆會在下文中🄵明。數值 `-1` 會🄵生「簡單🄵聲」，當無法播放其他音效時即作🄵最終退路。若系統回報錯誤，則會引發*RuntimeError*。

winsound.**SND_FILENAME**

sound 參數🄵 WAV 檔案名稱。請勿與*SND_ALIAS* 同時使用。

winsound.**SND_ALIAS**

sound 參數🄵登🄴檔中的音效關聯名稱。若登🄴檔中找不到此名稱，則會播放系統預設音效，除非同時指定了*SND_NODEFAULT*。若🄵有🄵🄵預設音效，則會引發*RuntimeError*。請勿與*SND_FILENAME*同時使用。

所有 Win32 系統至少支援以下項目；大多數系統支援更多：

PlaySound() name	對應的控制台音效名懲
'SystemAsterisk'	星號
'SystemExclamation'	驚嘆號
'SystemExit'	離開 Windows
'SystemHand'	關鍵性停止
'SystemQuestion'	問題

例如🄵：

```
import winsound
# 播放 Windows 的離開音效。
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# 若有🄵🄴預設音效，可能會播放該音效
# （因🄵 "*" 可能不是任何音效的🄵🄴名稱）。
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**

重🄵播放音效。必須同時使用*SND_ASYNC* 旗標以避免阻塞。不能與*SND_MEMORY* 一同使用。

winsound.**SND_MEMORY**

傳入*PlaySound()* 的 *sound* 參數是 WAV 檔案的記憶體映像，型🄵🄵*bytes-like object*。

> **備🄵**
>
> 此模組不支援從記憶體映像非同步播放音效，因此若同時使用本旗標與*SND_ASYNC*，將引發*RuntimeError*。

winsound.**SND_PURGE**

停止播放指定音效的所有實例。

> **備🄵**
>
> 此旗標在現代 Windows 平台上不支援。

winsound.**SND_ASYNC**

立即回傳，使音效可非同步播放。

winsound.**SND_NODEFAULT**

若找不到指定的音效，則不播放系統預設音效。

winsound.**SND_NOSTOP**

不中斷目前正在播放的音效。

winsound.**SND_NOWAIT**

若音效驅動程式正在忙碌，則立即回傳此。

> **備🄵**
>
> 此旗標在現代 Windows 平台上不支援。

winsound.**SND_APPLICATION**

sound 參數🄵登🄵檔中的應用程式特定🄵名。此旗標可與*SND_ALIAS* 一同使用，以指定應用程式自訂的音效🄵名。

winsound.**SND_SENTRY**

當音效被播放時，會觸發 SoundSentry 事件。

在 3.14 版被加入.

winsound.**SND_SYNC**

音效會同步播放。這是預設行🄵。

在 3.14 版被加入.

winsound.**SND_SYSTEM**

將音效指派給系統通知音效的音訊工作階段 (audio session)。

在 3.14 版被加入.

winsound.**MB_ICONASTERISK**

播放 `SystemDefault` 音效。

winsound.**MB_ICONEXCLAMATION**

播放 `SystemExclamation` 音效。

winsound.**MB_ICONHAND**

播放 `SystemHand` 音效。

winsound.**MB_ICONQUESTION**

播放 `SystemQuestion` 音效。

winsound.**MB_OK**

播放 `SystemDefault` 音效。

winsound.**MB_ICONERROR**

播放 `SystemHand` 音效。

在 3.14 版被加入.

winsound.**MB_ICONINFORMATION**

播放 `SystemDefault` 音效。

在 3.14 版被加入.

winsound.**MB_ICONSTOP**

播放 `SystemHand` 音效。

在 3.14 版被加入.

winsound.**MB_ICONWARNING**

播放 `SystemExclamation` 音效。

在 3.14 版被加入.

CHAPTER 34

Unix 特有服務

此章節所描述的模組 (module) 提供了針對 Unix 作業系統獨有特性的介面，或在某些情㠯下可用於其他 Unix 變形版本。以下㠯概述：

34.1 `shlex` --- 簡單的語法分析

原始碼： Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

shlex 模組定義了以下函式：

shlex.**split**(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

在 3.12 版的變更: Passing `None` for *s* argument now raises an exception, rather than reading *sys.stdin*.

shlex.**join**(*split_command*)

Concatenate the tokens of the list *split_command* and return a string. This function is the inverse of *split()*.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

The returned value is shell-escaped to protect against injection vulnerabilities (see *quote()*).

在 3.8 版被加入.

shlex.**quote**(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

> **⚠ 警告**
>
> The `shlex` module is **only designed for Unix shells**.
>
> The *`quote()`* function is not guaranteed to be correct on non-POSIX compliant shells or shells from other operating systems such as Windows. Executing commands quoted by this module on such shells can open up the possibility of a command injection vulnerability.
>
> Consider using functions that pass command arguments with lists such as *`subprocess.run()`* with `shell=False`.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *`split()`*:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

在 3.3 版被加入.

`shlex` 模組定義了以下類🄵:

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *`shlex`* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *`read()`* and *`readline()`* methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the *`infile`* attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *`shlex`* instance will operate in compatibility mode. When operating in POSIX mode, *`shlex`* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *`wordchars`* attribute that appear in *punctuation_chars* will be removed from *`wordchars`*. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon *`shlex`* instance creation and can't be modified later.

在 3.6 版的變更: 新增 *punctuation_chars* 參數。

也參考

configparser 模組
Parser for configuration files similar to the Windows `.ini` files.

34.1.1 shlex 物件

shlex 實例有以下方法：

shlex.**get_token**()

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.**push_token**(*str*)

Push the argument onto the token stack.

shlex.**read_token**()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.**sourcehook**(*filename*)

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.**pop_source**()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there. If *whitespace_split* is set to `True`, this will have no effect.

shlex.**whitespace**

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**

Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. When used in combination with *punctuation_chars*, tokens will be split on whitespace in addition to those characters.

在 3.8 版的變更: The *punctuation_chars* attribute was made compatible with the *whitespace_split* attribute.

shlex.**infile**

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**

The input stream from which this *shlex* instance is reading characters.

shlex.**source**

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**

If this attribute is numeric and `1` or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**

Source line number (count of newlines seen so far plus one).

shlex.**token**

The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**

Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.**punctuation_chars**

A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

在 3.6 版被加入.

34.1.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);
- If *whitespace_split* is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

34.1.3 Improved Compatibility with Shells

在 3.6 版被加入.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
```

(繼續下一頁)

(繼續上一頁)

```
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

備🄵

When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

However, to match the shell as closely as possible, it is recommended to always use `posix` and *whitespace_split* when using *punctuation_chars*, which will negate *wordchars* entirely.

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

34.2 `posix` --- 最常見的 POSIX 系統呼叫

該模組提供對由 C 標准和 POSIX 標准（一種🄵裝的 Unix 介面）所標准化的作業系統功能的存取。

可用性: Unix.

不要直接引入此模組。請改🄵引入*os* 模組，它提供了此介面的可移植 *(portable)* 版本。在 Unix 上，*os* 模組提供了*posix* 介面的超集 (superset)。在非 Unix 作業系統上，*posix* 模組不可用，但始終可以通過*os* 介面使用一個子集。一旦*os* 有被引入，使用它代替*posix* 不會有性能損失。此外，*os* 提供了一些額外的功能，例如當 `os.environ` 中的條目更改時自動呼叫*putenv()*。

錯誤會以例外的形式被回報；常見的例外是因🄵型🄵錯誤而給出的，而系統呼叫回報的錯誤會引發*OSError*。

34.2.1 對大檔案 (Large File) 的支援

一些作業系統（包括 AIX 和 Solaris）支援來自 C 程式模型且大於 2 GiB 的檔案，其中 `int` 和 `long` 是 32-bit（32 位元）的值。這通常透過將相關大小和偏移量 (offset) 種類定義🄵 64-bit 值來實作。此類檔案有時被稱🄵「大檔案 (*large files*)」。

當 `off_t` 的大小大於 `long` 且 `long long` 的大小至少與 `off_t` 相同時，對大檔案的支援會被🄴用。可能需要使用某些編譯器旗標來配置和編譯 Python 以🄴用此模式。例如，對於 Solaris 2.6 和 2.7，你需要執行如下操作：

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支援大檔案的 Linux 系統上，這可能有效：

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

34.2.2 值得注意的模組🄵容

除了 *os* 模組🄴明文件中描述的許多函式外，*posix* 還定義了以下資料項目：

posix.**environ**

表示直譯器🄵動時的字串環境的字典。鍵和值在 Unix 上是位元組，在 Windows 上是 str。例如，`environ[b'HOME']`（Windows 上🄵 `environ['HOME']`）是你的主目🄴的路徑名，等同於 C 語言中的 `getenv("HOME")`。

修改這個字典不會影響由 *execv()*、*popen()* 或 *system()* 傳遞的字串環境；如果你需要更改環境，請將 `environ` 傳遞給 *execve()* 或將變數賦值和匯出陳述句新增到 *system()* 或 *popen()* 的指令字串中。

在 3.2 版的變更: 在 Unix 上，鍵和值是位元組。

> **備🄵**
>
> *os* 模組提供了 `environ` 的替代實作，會在修改時更新環境。另請注意，更新 *os.environ* 將使該字典變成過時的。建議使用 *os* 模組版本，而不是直接存取 *posix* 模組。

34.3 pwd --- 密碼資料庫

此模組提供對 Unix 使用者帳🄵和密碼資料庫的存取介面。它適用於所有 Unix 版本。

可用性: Unix, not WASI, not iOS.

密碼資料庫條目被報告🄵類似元組的物件 (tuple-like object)，其屬性會對應於 `passwd` 結構的成員（屬性欄位請見下面的 `<pwd.h>`）：

索引	屬性	意義
0	`pw_name`	登🄵名
1	`pw_passwd`	可選的加密密碼
2	`pw_uid`	數值的使用者 ID
3	`pw_gid`	數值的群組 ID
4	`pw_gecos`	使用者名稱或🄴解欄位
5	`pw_dir`	使用者主目🄵 (home directory)
6	`pw_shell`	使用者命令直譯器

uid 和 gid 項目🄴整數，其他項目都是字串。如果找不到請求的條目，則會引發 *KeyError*。

備🄵

在傳統的 Unix 中，`pw_passwd` 欄位通常包含一個使用 DES 衍生演算法加密的密碼。然而，大多數現代 Unix 是使用所謂的 *shadow password* 系統。在那些 Unix 上，*pw_passwd* 欄位僅包含一個星號 (`'*'`) 或字母 `'x'`，其中加密密碼儲存在非全域可讀的 (not world readable) `/etc/shadow` 檔案中。*pw_passwd* 欄位是否包含任何有用的🄵容取🄵於系統。

它定義了以下項目：

`pwd.`**`getpwuid`**`(`*uid*`)`

回傳給定數值使用者 ID 的密碼資料庫條目。

`pwd.`**`getpwnam`**`(`*name*`)`

回傳給定使用者名稱的密碼資料庫條目。

`pwd.`**`getpwall`**`()`

以任意順序回傳所有可用密碼資料庫條目的 list。

也參考

***grp* 模組**
群組資料庫的介面，與此模組類似。

34.4 `grp` --- 群組資料庫

該模組提供對 Unix 群組資料庫的存取，所有 Unix 版本皆可使用。

可用性: Unix, not WASI, not Android, not iOS.

群組資料庫條目以類似 tuple 物件的方式報傳入，其屬性對應於 `group` 結構的成員（屬性欄位如下，請參🄵 `<grp.h>`）：

索引	屬性	含義
0	gr_name	群組名稱
1	gr_passwd	（被加密後的）群組密碼；大部分情🄵下是空的
2	gr_gid	群組的數字 ID
3	gr_mem	所有群組成員的使用者名稱

gid 是整數，name 和 password 是字串，而 member list 是字串串列。（請注意，根據密碼資料庫，大多數使用者🄵未被明確列🄵其所在群組的成員。請檢查這兩個資料庫，以取得完整的成員資訊。另外請注意，以 + 或 - 開頭的 `gr_name` 很可能是 YP/NIS 參照，可能無法透過*`getgrnam()`* 或*`getgrgid()`* 取得）。

它定義了以下項目：

`grp.`**`getgrgid`**`(`*id*`)`

回傳給定群組 ID 的群組資料庫條目。*`KeyError`* 會在找不到所要求的條目時被引發。

在 3.10 版的變更: *`TypeError`* 會在非整數的引數如浮點數或字串被傳入時被引發。

`grp.`**`getgrnam`**`(`*name*`)`

回傳給定群組名稱的群組資料庫條目。*`KeyError`* 會在找不到所要求的條目時引發。

grp.**getgrall**()

以任意順序傳回所有可用群組條目的串列。

也參考

pwd **模組**
使用者資料庫的介面，與此類似。

34.5 termios --- POSIX 風格 tty 控制

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see *termios(3)* Unix manual page. It is only available for those Unix versions that support POSIX *termios* style tty I/O control configured during installation.

可用性: Unix.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by sys.stdin.fileno(), or a *file object*, such as sys.stdin itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

此模組定義了以下函式:

termios.**tcgetattr**(*fd*)

Return a list containing the tty attributes for file descriptor *fd*, as follows: [iflag, oflag, cflag, lflag, ispeed, ospeed, cc] where *cc* is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the *cc* array must be done using the symbolic constants defined in the *termios* module.

termios.**tcsetattr**(*fd*, *when*, *attributes*)

Set the tty attributes for file descriptor *fd* from the *attributes*, which is a list like the one returned by *tcgetattr()*. The *when* argument determines when the attributes are changed:

termios.**TCSANOW**

Change attributes immediately.

termios.**TCSADRAIN**

Change attributes after transmitting all queued output.

termios.**TCSAFLUSH**

Change attributes after transmitting all queued output and discarding all queued input.

termios.**tcsendbreak**(*fd*, *duration*)

Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25--0.5 seconds; a nonzero *duration* has a system dependent meaning.

termios.**tcdrain**(*fd*)

Wait until all output written to file descriptor *fd* has been transmitted.

termios.**tcflush**(*fd*, *queue*)

Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

termios.**tcflow**(*fd*, *action*)

Suspend or resume input or output on file descriptor *fd*. The *action* argument can be `TCOOFF` to suspend output, `TCOON` to restart output, `TCIOFF` to suspend input, or `TCION` to restart input.

termios.**tcgetwinsize**(*fd*)

Return a tuple `(ws_row, ws_col)` containing the tty window size for file descriptor *fd*. Requires `termios.TIOCGWINSZ` or `termios.TIOCGSIZE`.

在 3.11 版被加入.

termios.**tcsetwinsize**(*fd*, *winsize*)

Set the tty window size for file descriptor *fd* from *winsize*, which is a two-item tuple `(ws_row, ws_col)` like the one returned by *tcgetwinsize()*. Requires at least one of the pairs (`termios.TIOCGWINSZ`, `termios.TIOCSWINSZ`); (`termios.TIOCGSIZE`, `termios.TIOCSSIZE`) to be defined.

在 3.11 版被加入.

也參考

***tty* 模組**
Convenience functions for common terminal control operations.

34.5.1 范例

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate *tcgetattr()* call and a `try` ... `finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

34.6 `tty` --- 終端機控制函式

原始碼： Lib/tty.py

---

tty 模組定義了將 tty 放入 cbreak 和 raw 模式的函式。

可用性: Unix.

因🄵它需要*termios* 模組，所以只能在 Unix 上執行。

tty 模組定義了以下函式：

tty.**cfmakeraw**(*mode*)

將 tty 屬性串列 *mode* 轉🄵成原始模式下的 tty 屬性串列，這個串列就像*termios.tcgetattr()* 所回傳的一樣。

在 3.12 版被加入.

tty.**cfmakecbreak**(*mode*)

將 tty 屬性串列 *mode* 轉🄵成 cbreak 模式下的 tty 屬性串列，這個串列就像*termios.tcgetattr()* 所回傳的一樣。

這會清除 *mode* 中的 ECHO 和 ICANON 本地模式旗標，🄵將最小輸入設定🄵 1 位元組，且無延遲。

在 3.12 版被加入.

在 3.12.2 版的變更: 不再清除 ICRNL 標記。這符合 Linux 和 macOS 的 `stty cbreak` 行🄵，也符合*setcbreak()* 歷來的做法。

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)

將檔案描述器 *fd* 的模式更改🄵 raw。如果 *when* 被省略，則預設🄵*termios.TCSAFLUSH*，🄵傳遞給*termios.tcsetattr()*。*termios.tcgetattr()* 的回傳值會在設定 *fd* 模式🄵 raw 之前先儲存起來。此函式回傳這個值。

在 3.12 版的變更: 現在的回傳值🄵原本的 tty 屬性，而不是 None。

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)

將檔案描述器 *fd* 的模式更改🄵 cbreak。如果 *when* 被省略，則預設🄵*termios.TCSAFLUSH*，🄵傳遞給*termios.tcsetattr()*。*termios.tcgetattr()* 的回傳值會在設定 *fd* 模式🄵 raw 之前先儲存起來。此函式回傳這個值。

這會清除 ECHO 和 ICANON 本地模式旗標，🄵將最小輸入設定🄵 1 位元組，且無延遲。

在 3.12 版的變更: 現在的回傳值🄵原本的 tty 屬性，而不是 None。

在 3.12.2 版的變更: 不再清除 ICRNL 旗標。這恢復了 Python 3.11 及更早版本的行🄵，也符合 Linux、macOS 及 BSD 在他們的 `stty(1)` man 頁面中描述的 cbreak 模式。

也參考

***termios* 模組**
低階終端機控制介面。

34.7 pty --- 🄵終端工具

原始碼： Lib/pty.py

The *pty* module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

可用性: Unix.

Pseudo-terminal handling is highly platform dependent. This code is mainly tested on Linux, FreeBSD, and macOS (it is supposed to work on other POSIX platforms but it's not been thoroughly tested).

pty 模組定義了以下函式：

pty.**fork**()

Fork. Connect the child's controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets *pid* 0, and the *fd* is *invalid*. The parent's return value is the *pid* of the child, and *fd* is a file descriptor connected to the child's controlling terminal (and also to the child's standard input and output).

警告

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using *urllib.request*.

pty.**openpty**()

Open a new pseudo-terminal pair, using *os.openpty()* if possible, or emulation code for generic Unix systems. Return a pair of file descriptors (master, slave), for the master and the slave end, respectively.

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])

Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal. It is expected that the process spawned behind the pty will eventually terminate, and when it does *spawn* will return.

A loop copies STDIN of the current process to the child and data received from the child to STDOUT of the current process. It is not signaled to the child if STDIN of the current process closes down.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an empty byte array should be returned to signal end of file.

The default implementation for both functions will read and return up to 1024 bytes each time the function is called. The *master_read* callback is passed the pseudoterminal's master file descriptor to read output from the child process, and *stdin_read* is passed file descriptor 0, to read from the parent process's standard input.

Returning an empty byte string from either callback is interpreted as an end-of-file (EOF) condition, and that callback will not be called after that. If *stdin_read* signals EOF the controlling terminal can no longer communicate with the parent process OR the child process. Unless the child process will quit without any input, *spawn* will then loop forever. If *master_read* signals EOF the same behavior results (on linux at least).

Return the exit status value from *os.waitpid()* on the child process.

os.waitstatus_to_exitcode() can be used to convert the exit status into an exit code.

引發一個附帶引數 argv 的稽核事件 pty.spawn。

在 3.4 版的變更: *spawn()* now returns the status value from *os.waitpid()* on the child process.

34.7.1 范例

The following program acts like the Unix command *script(1)*, using a pseudo-terminal to record all input and output of a terminal session in a "typescript".

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)
```

(繼續下一頁)

(繼續上一頁)

```
    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

34.8 fcntl --- fcntl 和 ioctl 系統呼叫

This module performs file and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. See the *fcntl(2)* and *ioctl(2)* Unix manual pages for full details.

可用性: Unix, not WASI.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an *io.IOBase* object, such as `sys.stdin` itself, which provides a *fileno()* that returns a genuine file descriptor.

在 3.3 版的變更: Operations in this module used to raise an *IOError* where they now raise an *OSError*.

在 3.8 版的變更: The `fcntl` module now contains `F_ADD_SEALS`, `F_GET_SEALS`, and `F_SEAL_*` constants for sealing of *os.memfd_create()* file descriptors.

在 3.9 版的變更: On macOS, the `fcntl` module exposes the `F_GETPATH` constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the `fcntl` module exposes the `F_OFD_GETLK`, `F_OFD_SETLK` and `F_OFD_SETLKW` constants, which are used when working with open file description locks.

在 3.10 版的變更: On Linux >= 2.6.11, the `fcntl` module exposes the `F_GETPIPE_SZ` and `F_SETPIPE_SZ` constants, which allow to check and modify a pipe's size respectively.

在 3.11 版的變更: On FreeBSD, the `fcntl` module exposes the `F_DUP2FD` and `F_DUP2FD_CLOEXEC` constants, which allow to duplicate a file descriptor, the latter setting `FD_CLOEXEC` flag in addition.

在 3.12 版的變更: On Linux >= 4.5, the *fcntl* module exposes the `FICLONE` and `FICLONERANGE` constants, which allow to share some data of one file with another file by reflinking on some filesystems (e.g., btrfs, OCFS2, and XFS). This behavior is commonly referred to as "copy-on-write".

在 3.13 版的變更: On Linux >= 2.6.32, the `fcntl` module exposes the `F_GETOWN_EX`, `F_SETOWN_EX`, `F_OWNER_TID`, `F_OWNER_PID`, `F_OWNER_PGRP` constants, which allow to direct I/O availability signals to a specific thread, process, or process group. On Linux >= 4.13, the `fcntl` module exposes the `F_GET_RW_HINT`, `F_SET_RW_HINT`, `F_GET_FILE_RW_HINT`, `F_SET_FILE_RW_HINT`, and `RWH_WRITE_LIFE_*` constants, which allow to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. On Linux >= 5.1 and NetBSD, the `fcntl` module exposes the `F_SEAL_FUTURE_WRITE` constant for use with `F_ADD_SEALS` and `F_GET_SEALS` operations. On FreeBSD, the `fcntl` module exposes the `F_READAHEAD`, `F_ISUNIONSTACK`, and `F_KINFO` constants. On macOS and FreeBSD, the `fcntl` module exposes the `F_RDAHEAD` constant. On NetBSD and AIX, the `fcntl` module exposes the `F_CLOSEM` constant. On NetBSD, the `fcntl` module exposes the `F_MAXFD` constant. On macOS and NetBSD, the `fcntl` module exposes the `F_GETNOSIGPIPE` and `F_SETNOSIGPIPE` constant.

在 3.14 版的變更: On Linux >= 6.1, the `fcntl` module exposes the `F_DUPFD_QUERY` to query a file descriptor pointing to the same file.

這個模組定義了以下函式:

`fcntl.`**`fcntl`**(*fd*, *cmd*, *arg=0*, */*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the *fcntl* module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, a *bytes-like object*, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

When *arg* is an integer, the function returns the integer return value of the C `fcntl()` call.

When the argument is bytes-like object, it represents a binary structure, for example, created by *struct.pack()*. A string value is encoded to binary using the UTF-8 encoding. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a *bytes* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes.

If the `fcntl()` call fails, an *OSError* is raised.

> **備🄵**
>
> If the type or the size of *arg* does not match the type or size of the argument of the operation (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating system is larger than 1024 bytes), this is most likely to result in a segmentation violation or a more subtle data corruption.

引發一個附帶引數 `fd`、`cmd`、`arg` 的稽核事件 `fcntl.fcntl`。

在 3.14 版的變更: Add support of arbitrary *bytes-like objects*, not only *bytes*.

`fcntl.`**`ioctl`**(*fd*, *request*, *arg=0*, *mutate_flag=True*, */*)

This function is identical to the *fcntl()* function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits or 64-bits, depending on the platform. Additional constants of interest for use as the *request* argument can be found in the *termios* module, under the same names as used in the relevant C header files.

The parameter *arg* can be an integer, a *bytes-like object*, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

If *arg* does not support the read-write buffer interface or the *mutate_flag* is false, behavior is as for the *fcntl()* function.

If *arg* supports the read-write buffer interface (like *bytearray*) and *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the `ioctl()`. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If the `ioctl()` call fails, an *OSError* exception is raised.

> **備🄵**
>
> If the type or size of *arg* does not match the type or size of the operation's argument (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating system is larger than 1024 bytes, or the size of the mutable bytes-like object is too small), this is most likely to result in a segmentation violation or a more subtle data corruption.

範例:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

引發一個附帶引數 `fd`、`request`、`arg` 的稽核事件 `fcntl.ioctl`。

在 3.14 版的變更: The GIL is always released during a system call. System calls failing with EINTR are automatically retried.

fcntl.**flock**(*fd*, *operation*, /)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` call fails, an *OSError* exception is raised.

引發一個附帶引數 `fd`、`operation` 的稽核事件 `fcntl.flock`。

fcntl.**lockf**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*, /)

This is essentially a wrapper around the *fcntl()* locking calls. *fd* is the file descriptor (file objects providing a *fileno()* method are accepted as well) of the file to lock or unlock, and *cmd* is one of the following values:

fcntl.**LOCK_UN**

Release an existing lock.

fcntl.**LOCK_SH**

Acquire a shared lock.

fcntl.**LOCK_EX**

Acquire an exclusive lock.

fcntl.**LOCK_NB**

Bitwise OR with any of the other three `LOCK_*` constants to make the request non-blocking.

If `LOCK_NB` is used and the lock cannot be acquired, an *OSError* will be raised and the exception will have an *errno* attribute set to *EACCES* or *EAGAIN* (depending on the operating system; for portability, check for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with *io.IOBase.seek()*, specifically:

- `0` -- relative to the start of the file (*os.SEEK_SET*)
- `1` -- relative to the current buffer position (*os.SEEK_CUR*)
- `2` -- relative to the end of the file (*os.SEEK_END*)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

引發一個附帶引數 `fd`、`cmd`、`len`、`start`、`whence` 的稽核事件 `fcntl.lockf`。

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a *bytes* object. The structure lay-out for the *lockdata* variable is system dependent --- therefore using the *flock()* call may be better.

也參考

os 模組
If the locking flags *O_SHLOCK* and *O_EXLOCK* are present in the *os* module (on BSD only), the *os.open()* function provides an alternative to the *lockf()* and *flock()* functions.

34.9 resource --- 資源使用資訊

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

可用性: Unix, not WASI.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An *OSError* is raised on syscall failure.

exception resource.**error**
A deprecated alias of *OSError*.

在 3.3 版的變更: Following **PEP 3151**, this class was made an alias of *OSError*.

34.9.1 Resource Limits

Resources usage can be limited using the *setrlimit()* function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the *getrlimit(2)* man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.**RLIM_INFINITY**
Constant used to represent the limit for an unlimited resource.

resource.**getrlimit**(*resource*)
Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises *ValueError* if an invalid resource is specified, or *error* if the underlying system call fails unexpectedly.

resource.**setrlimit**(*resource*, *limits*)
Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of *RLIM_INFINITY* can be used to request a limit that is unlimited.

Raises *ValueError* if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of *RLIM_INFINITY* when the hard or system limit for that resource is not unlimited will result in a *ValueError*. A process with the effective UID of super-user can request any valid limit value, including unlimited, but *ValueError* will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise *error* if the underlying system call fails.

VxWorks only supports setting *RLIMIT_NOFILE*.

引發一個附帶引數 `resource`、`limits` 的稽核事件 `resource.setrlimit`。

resource.**prlimit**(*pid*, *resource*[, *limits*])
Combines *setrlimit()* and *getrlimit()* in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in *setrlimit()*, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises *ProcessLookupError* when *pid* can't be found and *PermissionError* when the user doesn't have CAP_SYS_RESOURCE for the process.

引發一個附帶引數 pid、resource、limits 的稽核事件 resource.prlimit。

可用性: Linux >= 2.6.36 with glibc >= 2.13.

在 3.4 版被加入.

These symbols define resources whose consumption can be controlled using the *setrlimit()* and *getrlimit()* functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences --- symbols not defined for a platform will not be available from this module on that platform.

resource.**RLIMIT_CORE**

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.**RLIMIT_CPU**

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a SIGXCPU signal is sent to the process. (See the *signal* module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.**RLIMIT_FSIZE**

The maximum size of a file which the process may create.

resource.**RLIMIT_DATA**

The maximum size (in bytes) of the process's heap.

resource.**RLIMIT_STACK**

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource.**RLIMIT_RSS**

The maximum resident set size that should be made available to the process.

resource.**RLIMIT_NPROC**

The maximum number of processes the current process may create.

resource.**RLIMIT_NOFILE**

The maximum number of open file descriptors for the current process.

resource.**RLIMIT_OFILE**

The BSD name for *RLIMIT_NOFILE*.

resource.**RLIMIT_MEMLOCK**

The maximum address space which may be locked in memory.

resource.**RLIMIT_VMEM**

The largest area of mapped memory which the process may occupy.

可用性: FreeBSD >= 11.

resource.**RLIMIT_AS**

The maximum area (in bytes) of address space which may be taken by the process.

resource.**RLIMIT_MSGQUEUE**

The number of bytes that can be allocated for POSIX message queues.

可用性: Linux >= 2.6.8.

在 3.4 版被加入.

resource.**RLIMIT_NICE**

The ceiling for the process's nice level (calculated as 20 - rlim_cur).

可用性: Linux >= 2.6.12.

在 3.4 版被加入.

resource.**RLIMIT_RTPRIO**

The ceiling of the real-time priority.

可用性: Linux >= 2.6.12.

在 3.4 版被加入.

resource.**RLIMIT_RTTIME**

The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

可用性: Linux >= 2.6.25.

在 3.4 版被加入.

resource.**RLIMIT_SIGPENDING**

The number of signals which the process may queue.

可用性: Linux >= 2.6.8.

在 3.4 版被加入.

resource.**RLIMIT_SBSIZE**

The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

可用性: FreeBSD.

在 3.4 版被加入.

resource.**RLIMIT_SWAP**

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see tuning(7) for a complete description of this sysctl.

可用性: FreeBSD.

在 3.4 版被加入.

resource.**RLIMIT_NPTS**

The maximum number of pseudo-terminals created by this user id.

可用性: FreeBSD.

在 3.4 版被加入.

resource.**RLIMIT_KQUEUES**

The maximum number of kqueues this user id is allowed to create.

可用性: FreeBSD >= 11.

在 3.10 版被加入.

34.9.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.**getrusage**(*who*)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_* constants described below.

一個簡單範例:

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating-point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here:

Index	Field	Resource
0	ru_utime	time in user mode (float seconds)
1	ru_stime	time in system mode (float seconds)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

resource.**getpagesize**()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the *getrusage()* function to specify which processes information should be provided for.

resource.**RUSAGE_SELF**

Pass to *getrusage()* to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

resource.**RUSAGE_CHILDREN**

Pass to *getrusage()* to request resources consumed by child processes of the calling process which have been terminated and waited for.

resource.**RUSAGE_BOTH**

Pass to *getrusage()* to request resources consumed by both the current process and child processes. May not be available on all systems.

resource.**RUSAGE_THREAD**

Pass to *getrusage()* to request resources consumed by the current thread. May not be available on all systems.

在 3.2 版被加入.

34.10 `syslog` --- Unix syslog 函式庫例程

This module provides an interface to the Unix `syslog` library routines. Refer to the Unix manual pages for a detailed description of the `syslog` facility.

可用性: Unix, not WASI, not iOS.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the *logging.handlers* module as *SysLogHandler*.

該模組定義了以下函式:

syslog.**syslog**(*message*)

syslog.**syslog**(*priority*, *message*)

Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to *LOG_INFO*, determines the message priority. If the facility is not encoded in *priority* using logical-or (`LOG_INFO | LOG_USER`), the value given in the *openlog()* call is used.

If *openlog()* has not been called prior to the call to *syslog()*, *openlog()* will be called with no arguments.

引發一個附帶引數 `priority`、`message` 的稽核事件 `syslog.syslog`。

在 3.2 版的變更: In previous versions, *openlog()* would not be called automatically if it wasn't called prior to the call to *syslog()*, deferring to the syslog implementation to call `openlog()`.

在 3.12 版的變更: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) *openlog()* must be called in the main interpreter before *syslog()* may be used in a subinterpreter. Otherwise it will raise *RuntimeError*.

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])

Logging options of subsequent *syslog()* calls can be set by calling *openlog()*. *syslog()* will call *openlog()* with no arguments if the log is not currently open.

The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field -- see below for possible values to combine. The optional *facility* keyword argument (default is *LOG_USER*) sets the default facility for messages which do not have a facility explicitly encoded.

引發一個附帶引數 `ident`、`logoption`、`facility` 的稽核事件 `syslog.openlog`。

在 3.2 版的變更: In previous versions, keyword arguments were not allowed, and *ident* was required.

在 3.12 版的變更: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise *RuntimeError* if called in a subinterpreter.

syslog.**closelog**()

Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, *openlog()* will be called on the first *syslog()* call (if *openlog()* hasn't already been called), and *ident* and other *openlog()* parameters are reset to defaults.

引發一個不附帶引數的稽核事件 `syslog.closelog`。

在 3.12 版的變更: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise *RuntimeError* if called in a subinterpreter.

syslog.**setlogmask**(*maskpri*)

Set the priority mask to *maskpri* and return the previous mask value. Calls to *syslog()* with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

引發一個附帶引數 `maskpri` 的稽核事件 `syslog.setlogmask`。

這個模組定義了以下常數:

syslog.**LOG_EMERG**

syslog.**LOG_ALERT**

syslog.**LOG_CRIT**

syslog.**LOG_ERR**

syslog.**LOG_WARNING**

syslog.**LOG_NOTICE**

syslog.**LOG_INFO**

syslog.**LOG_DEBUG**

Priority levels (high to low).

syslog.**LOG_AUTH**

syslog.**LOG_AUTHPRIV**

syslog.**LOG_CRON**

syslog.**LOG_DAEMON**

syslog.**LOG_FTP**

syslog.**LOG_INSTALL**

syslog.**LOG_KERN**

syslog.**LOG_LAUNCHD**

syslog.**LOG_LPR**

syslog.**LOG_MAIL**

syslog.**LOG_NETINFO**

syslog.**LOG_NEWS**

syslog.**LOG_RAS**

syslog.**LOG_REMOTEAUTH**

syslog.**LOG_SYSLOG**

syslog.**LOG_USER**

syslog.**LOG_UUCP**

syslog.**LOG_LOCAL0**

syslog.**LOG_LOCAL1**

syslog.**LOG_LOCAL2**

syslog.**LOG_LOCAL3**

syslog.**LOG_LOCAL4**

syslog.**LOG_LOCAL5**

syslog.**LOG_LOCAL6**

syslog.**LOG_LOCAL7**

Facilities, depending on availability in <syslog.h> for *LOG_AUTHPRIV*, *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL* and *LOG_RAS*.

在 3.13 版的變更: Added *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL*, *LOG_RAS*, and *LOG_LAUNCHD*.

syslog.**LOG_PID**

syslog.**LOG_CONS**

syslog.**LOG_NDELAY**

syslog.**LOG_ODELAY**

syslog.**LOG_NOWAIT**

syslog.**LOG_PERROR**

Log options, depending on availability in <syslog.h> for *LOG_ODELAY*, *LOG_NOWAIT* and *LOG_PERROR*.

34.10.1 范例

簡單范例

一組簡單範例:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER 35

模組命令列介面

以下模組具有命令列介面。

- *ast*
- *asyncio*
- `base64`
- *calendar*
- `code`
- *compileall*
- `cProfile`: 請見*profile*
- *dis*
- *doctest*
- `encodings.rot_13`
- `ensurepip`
- `filecmp`
- `fileinput`
- `ftplib`
- *gzip*
- *http.server*
- `idlelib`
- *inspect*
- *json*
- *mimetypes*
- `pdb`
- *pickle*
- *pickletools*

- *platform*
- `poplib`
- *profile*
- `pstats`
- *py_compile*
- `pyclbr`
- `pydoc`
- `quopri`
- *random*
- `runpy`
- *site*
- *sqlite3*
- *symtable*
- *sysconfig*
- `tabnanny`
- *tarfile*
- `this`
- *timeit*
- *tokenize*
- *trace*
- `turtledemo`
- *unittest*
- *uuid*
- `venv`
- `webbrowser`
- *zipapp*
- *zipfile*

另請見 Python 命令列介面。

CHAPTER 36

已被取代的模組

此章節所描述的模組在大多數使用情[F]下已被其他模組取代，僅[F]了向後相容性而被保留下來。

某些模組出現在本章節，可能是因[F]它們能解[F]的問題是問題空間的有限子集，而標准函式庫的其他地方提供了更通用的解[F]方案（例如，*getopt* 只處理「在 Python 中模擬 C 語言中 `getopt()` 這個 API」這項非常具體的任務，而非像*optparse* 和*argparse* 那樣提供更廣泛的命令列選項剖析與引數剖析功能）。

或者，某些模組出現在本章節，可能是因[F]它們已被明確[F]用（deprecated），[F]將在未來版本中移除；又或者它們屬於*soft deprecated* 狀態，官方[F]烈不建議在新專案中使用。隨著 **PEP 594** 移除了多個過時模組，目前標准函式庫中已不存在後者（軟性[F]用）這類模組。

36.1 `getopt` --- 用於命令列選項的 C 風格剖析器

原始碼： Lib/getopt.py

備[F]

This module is considered feature complete. A more declarative and extensible alternative to this API is provided in the *optparse* module. Further functional enhancements for command line parameter processing are provided either as third party modules on PyPI, or else as features in the *argparse* module.

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the Unix `getopt()` function (including the special meanings of arguments of the form '-' and '--'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

Users who are unfamiliar with the Unix `getopt()` function should consider using the *argparse* module instead. Users who are familiar with the Unix `getopt()` function, but would like to get equivalent behavior while writing less code and getting better help and error messages should consider using the *optparse* module. See 選擇一個命令列參數剖析函式庫 for additional details.

這個模組提供兩個函式和一個例外：

`getopt.`**`getopt`**(*args*, *shortopts*, *longopts=[]*)

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *shortopts* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`)

and options that accept an optional argument followed by two colons (`'::'`); i.e., the same format that Unix `getopt()` uses.

> **備🄵**
>
> Unlike GNU `getopt()`, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Long options which accept an optional argument should be followed by an equal sign and question mark (`'=?'`). To accept only long options, *shortopts* should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if *longopts* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so *GetoptError* will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of *args*). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `'-x'`) or two hyphens for long options (e.g., `'--long-option'`), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

在 3.14 版的變更: Optional arguments are supported.

`getopt.`**`gnu_getopt`**(*args*, *shortopts*, *longopts=[]*)

This function works like *getopt()*, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The *getopt()* function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is `'+'`, or if the environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

If the first character of the option string is `'-'`, non-option arguments that are followed by options are added to the list of option-and-value pairs as a pair that has `None` as its first element and the list of non-option arguments as its second element. The second element of the `gnu_getopt()` result is a list of program arguments after the last option.

在 3.14 版的變更: Support for returning intermixed options and non-option arguments in order.

exception `getopt.`**`GetoptError`**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

exception `getopt.`**`error`**

🄵了向後相容性而設的*GetoptError* 🄵名。

一個僅使用 Unix 風格選項的範例:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

使用長選項名稱同樣容易：

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

Optional arguments should be specified explicitly:

```
>>> s = '-Con -C --color=off --color a1 a2'
>>> args = s.split()
>>> args
['-Con', '-C', '--color=off', '--color', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'C::', ['color=?'])
>>> optlist
[('-C', 'on'), ('-C', ''), ('--color', 'off'), ('--color', '')]
>>> args
['a1', 'a2']
```

The order of options and non-option arguments can be preserved:

```
>>> s = 'a1 -x a2 a3 a4 --long a5 a6'
>>> args = s.split()
>>> args
['a1', '-x', 'a2', 'a3', 'a4', '--long', 'a5', 'a6']
>>> optlist, args = getopt.gnu_getopt(args, '-x:', ['long='])
>>> optlist
[(None, ['a1']), ('-x', 'a2'), (None, ['a3', 'a4']), ('--long', 'a5')]
>>> args
['a6']
```

在🄵本中，典型的用法如下：

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # 印出幫助訊息🄵退出：
        print(err)  # 會印出像是 "option -a not recognized" 的訊息
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    process(args, output=output, verbose=verbose)
```

(繼續下一頁)

(繼續上一頁)

```
if __name__ == "__main__":
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the *optparse* module:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

A roughly equivalent command line interface for this case can also be produced by using the *argparse* module:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

參見*選擇一個命令列參數剖析函式庫* 以🄵解這段程式碼的 `argparse` 版本與 `optparse`（以及 `getopt`）版本在行🄵上的差🄵。

也參考

***optparse* 模組**
宣告式命令列選項剖析。

***argparse* 模組**
More opinionated command line option and argument parsing library.

CHAPTER 37

已移除的模組

本章描述的模組已從 Python 標准函式庫中移除。這🄵提供文件以協助尋找替代方案。

37.1 `aifc` --- 讀寫 AIFF 與 AIFC 檔案

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 `aifc` 模組的 Python 版本是 Python 3.12。

37.2 `asynchat` --- 非同步 socket 指令/回應處理函式

Deprecated since version 3.6, removed in version 3.12.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.6 中被🄵用，🄵已在 Python 3.12 中被移除。它的移除是在 **PEP 594** 中🄵定的。

應用程式應改用*asyncio* 模組。

最後提供 `asynchat` 模組的 Python 版本是 Python 3.11。

37.3 `asyncore` --- 非同步 socket 處理函式

Deprecated since version 3.6, removed in version 3.12.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.6 中被🄵用，🄵已在 Python 3.12 中被移除。它的移除是在 **PEP 594** 中🄵定的。

應用程式應改用*asyncio* 模組。

最後提供 `asyncore` 模組的 Python 版本是 Python 3.11。

37.4 audioop --- 操作原始聲音檔案

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 audioop 模組的 Python 版本是 Python 3.12。

37.5 cgi --- 通用閘道器介面支援

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

可以改用 PyPI 上的模組 fork：legacy-cgi。這是 cgi 模組的一個副本，不再由 Python 核心團隊維護或支援。

最後提供 cgi 模組的 Python 版本是 Python 3.12。

37.6 cgitb --- CGI 🄵本的回溯管理器 (traceback manager)

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

可以改用 PyPI 上的模組 fork：legacy-cgi。這是 cgi 模組的一個🄵本，不再由 Python 核心團隊維護或支援。

最後提供 cgitb 模組的 Python 版本是 Python 3.12。

37.7 chunk --- 讀取 IFF 分塊資料

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 chunk 模組的 Python 版本是 Python 3.12。

37.8 crypt --- 用於檢查 Unix 密碼的函式

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

應用程式可以改用標准函式庫中的*hashlib*模組。其他可能的替代方案是 PyPI 上的第三方庫：legacycrypt、bcrypt、argon2-cffi 或 passlib。這些函式庫🄵不受 Python 核心團隊支援或維護。

最後提供 crypt 模組的 Python 版本是 Python 3.12。

37.9 distutils --- 建置與安裝 Python 模組

Deprecated since version 3.10, removed in version 3.12.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.10 中被🄵用，🄵已在 Python 3.12 中被移除。它的移除是在 **PEP 632** 中🄵定的，該 PEP 附有遷移建議。

最後提供 distutils 模組的 Python 版本是 Python 3.11。

37.10 imghdr --- 判定圖片種類

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄴定的。

可能的替代方案是 PyPI 上的第三方函式庫：filetype、puremagic 或 python-magic。它們🄵不受 Python 核心團隊支援或維護。

最後提供 imghdr 模組的 Python 版本是 Python 3.12。

37.11 imp --- 存取引入系統🄵層

Deprecated since version 3.4, removed in version 3.12.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.4 中被🄵用，🄵已在 Python 3.12 中被移除。

其移除通知包含了從 imp 遷移至*importlib* 的指引。

最後提供 imp 模組的 Python 版本是 Python 3.11。

37.12 mailcap --- Mailcap 檔案處理

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄴定的。

最後提供 mailcap 模組的 Python 版本是 Python 3.12。

37.13 msilib --- 讀寫 Microsoft Installer 檔案

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄴定的。

最後提供 msilib 模組的 Python 版本是 Python 3.12。

37.14 nis --- Sun NIS (Yellow Pages) 介面

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄴定的。

最後提供 nis 模組的 Python 版本是 Python 3.12。

37.15 nntplib --- NNTP 協定用🄵端

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄴定的。

最後提供 nntplib 模組的 Python 版本是 Python 3.12。

37.16 ossaudiodev --- 對 OSS 相容聲音裝置的存取

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 ossaudiodev 模組的 Python 版本是 Python 3.12。

37.17 pipes --- shell pipelines 介面

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

應用程式應改用 *subprocess* 模組。

最後提供 pipes 模組的 Python 版本是 Python 3.12。

37.18 smtpd --- SMTP 伺服器

Deprecated since version 3.6, removed in version 3.12.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.6 中被🄵用，🄵已在 Python 3.12 中被移除。它的移除是在 **PEP 594** 中🄵定的。

一個可能的替代方案是第三方 aiosmtpd 函式庫。這個函式庫🄵不是由 Python 核心團隊維護或支援。

最後提供 smtpd 模組的 Python 版本是 Python 3.11。

37.19 sndhdr --- 判定聲音檔案的種類

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

可能的替代方案是 PyPI 上的第三方模組：filetype、puremagic 或 python-magic。它們🄵不受 Python 核心團隊支援或維護。

最後提供 sndhdr 模組的 Python 版本是 Python 3.12。

37.20 spwd --- shadow 密碼資料庫

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

可能的替代方案是 PyPI 上的第三方函式庫：python-pam。這🄵不受 Python 核心團隊支援或維護。

最後提供 spwd 模組的 Python 版本是 Python 3.12。

37.21 sunau --- 讀寫 Sun AU 檔案

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 sunau 模組的 Python 版本是 Python 3.12。

37.22 `telnetlib` --- Telnet 用🄵端

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

可能的替代方案是 PyPI 上的第三方函式庫：telnetlib3 或 Exscript。它們🄵不受 Python 核心團隊支援或維護。

最後提供 `telnetlib` 模組的 Python 版本是 Python 3.12。

37.23 `uu` --- uuencode 檔案的編碼與解碼

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 `uu` 模組的 Python 版本是 Python 3.12。

37.24 `xdrlib` --- XDR 資料的編碼與解碼

Deprecated since version 3.11, removed in version 3.13.

這個模組已不再是 Python 標准函式庫的一部分。它在 Python 3.11 中被🄵用，🄵已在 Python 3.13 中被移除。它的移除是在 **PEP 594** 中🄵定的。

最後提供 `xdrlib` 模組的 Python 版本是 Python 3.12。

CHAPTER 38

安全性注意事項

以下模組具有特定的安全性注意事項：

- *base64*：*base64* 安全性注意事項在 **RFC 4648**
- *hashlib*：所有建構函式都用 *"usedforsecurity"* 僅限關鍵字引數，禁用已知的不安全與被阻擋的演算法
- *http.server* 不適合在正式環境使用，因其僅實作基本的安全檢查。請參安全注意事項。
- *logging*：日配置使用 *eval()*
- *multiprocessing*：*Connection.recv()* 使用 *pickle*
- *pickle*：限制 *pickle* 中的全域變數
- *random* 不該用於安全性相關用途，請改用 *secrets*
- *shelve*：*shelve* 基於 *pickle*，因此不適合用來處理不受信任的來源
- *ssl*：*SSL/TLS* 安全性注意事項
- *subprocess*：子行程安全性注意事項
- *tempfile*：*mktemp* 由於存在競條件 *(race condition)* 漏洞而被用
- *xml*：*XML* 安全性
- *zipfile*：惡意準備的 *.zip* 檔案可能會導致硬碟空間耗盡

`-I` 命令列選項可用於在隔離模式下運行 Python。若其無法使用，可以改用 `-P` 選項或 `PYTHONSAFEPATH` 環境變數，避免 *sys.path* 新增在的不安全路徑，例如目前目、本的目或空字串。

APPENDIX A

術語表

>>>
互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...
可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）🄵部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，互動式 shell 顯示的預設 Python 提示字元。
- The three dots form of the *Ellipsis* object.

abstract base class（抽象基底類🄵）
抽象基底類🄵（又稱🄵 ABC）提供了一種定義介面的方法，作🄵*duck-typing*（鴨子型🄵）的補充。其他類似的技術，像是`hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC 🄵用🄵擬的 subclass（子類🄵），它們🄵不繼承自另一個 class（類🄵），但仍可被`isinstance()` 及`issubclass()` 辨識；請參🄵`abc` 模組的🄵明文件。Python 有許多🄵建的 ABC，用於資料結構（在`collections.abc` 模組）、數字（在`numbers` 模組）、串流（在`io` 模組）及 import 尋檢器和載入器（在`importlib.abc` 模組）。你可以使用`abc` 模組建立自己的 ABC。

annotate function（🄵釋函式）
A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation（🄵釋）
一個與變數、class 屬性、函式的參數或回傳值相關聯的標🄵。照慣例，它被用來作🄵*type hint*（型🄵提示）。

在 runtime 的區域變數🄵釋無法被存取，但全域變數、類🄵屬性和函式的🄵釋，分🄵能🄵以對模組、類🄵和函式呼叫`annotationlib.get_annotations()` 來取得。

請參🄵*variable annotation*、*function annotation*、**PEP 484**、**PEP 526** 和 **PEP 649**，這些章節皆有此功能的🄵明。關於🄵釋的最佳實踐方法也請參🄵 annotations-howto。

argument（引數）
呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識🄵字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）🄵的值被傳遞的引數。例如，`3` 和 `5` 都是以下`complex()` 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置引數 (positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作🄵 `*` 之後的*iterable*（可🄵代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

  ```
  complex(3, 5)
  complex(*(3, 5))
  ```

  引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參🄵 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

  另請參🄵術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差🄵，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）
一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步🄵生器）
一個會回傳*asynchronous generator iterator*（非同步🄵生器🄵代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 🄵圈的值。

這個術語通常用來表示一個非同步🄵生器函式，但在某些情境中，也可能是表示非同步🄵生器🄵代器 *(asynchronous generator iterator)*。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

一個非同步🄵生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步🄵生器🄵代器）
一個由*asynchronous generator*（非同步🄵生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步🄵代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步🄵生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，🄵記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步🄵生器🄵代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參🄵 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可🄵代物件）
一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步🄵代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步🄵代器）
一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__()` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步🄵代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute（屬性）
一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

如果一個物件允許，給予該物件一個名稱不是由 identifiers 所定義之識🄴符 (identifier) 的屬性是有可能的，例如使用`setattr()`。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用`getattr()` 來取得它。

awaitable（可等待物件）
一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參🄵 **PEP 492**。

BDFL
Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進位檔案）
一個能🄴讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進位檔案的例子有：以二進位模式（`'rb'`、`'wb'` 或 `'rb+'`）開🄴的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及`io.BytesIO` 和`gzip.GzipFile` 實例。

另請參🄵*text file*（文字檔案），它是一個能🄴讀取和寫入`str` 物件的檔案物件。

borrowed reference（借用參照）
在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼🄵不擁有這個參照。如果該物件被銷🄴，它會成🄵一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（🄵參照），而將該物件銷🄴。

對*borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉🄵🄵*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷🄴。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）
一個支援 bufferobjects 且能🄴匯出 C-*contiguous* 緩衝區的物件。這包括所有的`bytes`、`bytearray` 和`array.array` 物件，以及許多常見的`memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。🄴明文件通常會將這些物件稱🄵「可讀寫的類位元組串物件」。可變緩衝區的物件包括`bytearray`，以及`bytearray` 的`memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括`bytes`，以及`bytes` 物件的`memoryview`。

bytecode（位元組碼）
Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的🄵部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能🄵更快速（可以不用從原始碼重新編譯🄵位元組碼）。這種「中間語言 (intermediate language)」據🄵是運行在一個*virtual machine*（🄵擬機器）上，該🄵擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 🄵擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在*dis* 模組的🄵明文件中找到。

callable（可呼叫物件）
一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

```
callable(argument1, argument2, argumentN)
```

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼）
作🄵引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類🄵）
一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類🄵變數）
一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

closure variable（閉包變數）
從外部作用域中定義且從巢狀作用域參照的自由變數，不是於 runtime 從全域或🄵建命名空間解析。可以使用 `nonlocal` 關鍵字明確定義以允許寫入存取，或者如果僅需讀取變數則隱式定義即可。

例如在下面程式碼中的 `inner` 函式中，`x` 和 `print` 都是自由變數，但只有 `x` 是閉包變數：

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由於 `codeobject.co_freevars` 屬性（🄵管名稱如此，但它僅包含閉包變數的名稱，而不是列出所有參照的自由變數），當預期含義是特指閉包變數時，有時候甚至也會使用更通用的自由變數一詞。

complex number（🄵數）
一個我們熟悉的實數系統的擴充，在此所有數字都會被表示🄵一個實部和一個🄵部之和。🄵數就是🄵數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫🄵 `i`，在工程學中被寫🄵 `j`。Python 🄵建了對🄵數的支援，它是用後者的記法來表示🄵數；🄵部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將*`math`* 模組🄵的工具等效地用於🄵數，請使用*`cmath`* 模組。🄵數的使用是一個相當進階的數學功能。如果你🄵有察覺到對它們的需求，那🄵幾乎能確定你可以安全地忽略它們。

context（情境）
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular *`contextvars.Context`* object and accessed via *`ContextVar`* objects. Also see *context variable*.
- 一個*`contextvars.Context`* 物件。另請參🄵*current context*。

context management protocol（情境管理協定）
由 `with` 陳述式所呼叫的 `__enter__()` 和 `__exit__()` 方法。另請參🄵 **PEP 343**。

context manager（情境管理器）
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

context variable（情境變數）
A variable whose value depends on which context is the *current context*. Values are accessed via *`contextvars.ContextVar`* objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous（連續的）
如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視🄵是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序🄵覽各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程）
協程是副程式 (subroutine) 的一種更🄵廣義的形式。副程式是在某個時間點被進入🄵在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能🄵以 `async def` 陳述式被實作。另請參🄵 **PEP 492**。

coroutine function（協程函式）
一個回傳*coroutine*（協程）物件的函式。一個協程函式能以 `async def` 陳述式被定義，🄵可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython
Python 程式語言的標准實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

current context
The *context* (*contextvars.Context* object) that is currently used by *ContextVar* objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see *asyncio*) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

decorator（裝飾器）
一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用🄵一種函式的變🄵 (function transformation)。裝飾器的常見範例是*classmethod()* 和*staticmethod()*。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Class 也存在相同的概念，但在那🄵比較不常用。關於裝飾器的更多🄵容，請參🄵函式定義和 class 定義的🄵明文件。

descriptor（描述器）
任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行🄵會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或🄵除某個屬性時，會在 *a* 的 class 字典中查找名稱🄵 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因🄵它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、🄵態 method，以及對 super class（父類🄵）的參照。

關於描述器 method 的更多資訊，請參🄵 descriptors 或描述器使用指南。

dictionary（字典）
一個關聯陣列 (associative array)，其中任意的鍵會被對映到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱🄵雜🄵 (hash)。

dictionary comprehension（字典綜合運算）
一種緊密的方法，用來處理一個可🄵代物件中的全部或部分元素，🄵將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會🄵生一個字典，它包含了鍵 `n` 對映到值 `n ** 2`。請參🄵 comprehensions。

dictionary view（字典檢視）
從*dict.keys()*、*dict.values()* 及*dict.items()* 回傳的物件被稱🄵字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要🄵制將字典檢視轉🄵完整的 list（串列），須使用 `list(dictview)`。請參🄵字典視圖物件。

docstring（🄵明字串）
一個在 class、函式或模組中，作🄵第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，🄵被放入所屬 class、函式或模組的*__doc__* 屬性中。由於🄵明字串可以透過🄵省 (introspection) 來🄵覽，因此它是物件的🄵明文件存放的標准位置。

duck-typing（鴨子型🄵）
一種程式設計風格，它不是藉由檢查一個物件的型🄵來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一🄵鴨子而且叫起來像一🄵鴨子，那🄵它一定是一🄵鴨子。」）因🄵🄵調介面而非特定型🄵，精心設計的程式碼能讓多形替代 (polymorphic

substitution) 來增進它的靈活性。鴨子型🄵要避免使用*type()* 或*isinstance()* 進行測試。（但是請注意，鴨子型🄵可以用抽象基底類🄵 *(abstract base class)* 來補充。）然而，它通常會🄵用*hasattr()* 測試，或是*EAFP* 程式設計風格。

dunder（雙底🄵）
一個非正式的縮寫，代表「雙底🄵 (double underscore)」，當談論到特殊方法時使用。例如，`__init__` 通常被叫做"dunder init"。

EAFP
Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，🄵在該假設被推翻時再捕獲例外。這種乾🄵且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

evaluate function（求值函式）
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression（運算式）
一段可以被評估🄴求值的語法。🄵句話🄵，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，🄵非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組）
一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）
f-strings（f 字串）
以 `f` 或 `F` 🄵前綴的字串文本通常被稱🄵「f 字串」，它是格式化的字串文本的縮寫。另請參🄵 **PEP 498**。

file object（檔案物件）
一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能🄵協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標準輸入 / 輸出、記憶體🄵緩衝區、socket（插座）、管🄵 (pipe) 等）的存取。檔案物件也被稱🄵類檔案物件 *(file-like object)* 或串流 *(stream)*。

實際上，有三種檔案物件：原始的二進位檔案、緩衝的二進位檔案和文字檔案。它們的介面在*io* 模組中被定義。建立檔案物件的標準方法是使用*open()* 函式。

file-like object（類檔案物件）
file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）
Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發*UnicodeError*。

sys.getfilesystemencoding() 和*sys.getfilesystemencodeerrors()* 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python 🄴動時由 `PyConfig_Read()` 函式來配置：請參🄵 `filesystem_encoding`，以及 `PyConfig` 的成員 `filesystem_errors`。

另請參🄵*locale encoding*（區域編碼）。

finder（尋檢器）
一個物件，它會嘗試🄵正在被 import 的模組尋找*loader*（載入器）。

有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用*sys.meta_path*，而*路徑項目尋檢器 (path entry finder)* 會使用*sys.path_hooks*。

請參🄵 finders-and-loaders 和*importlib* 以了解更多細節。

floor division（向下取整除法）
向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果🄵 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因🄵是 `-2.75` 被向下無條件舍去。請參🄵 **PEP 238**。

free threading（自由執行緒）
🄵一種執行緒模型，多個執行緒可以在同一直譯器中同時運行 Python 位元組碼。這與全域直譯器鎖形成對比，後者一次只允許一個執行緒執行 Python 位元組碼。請參🄵 **PEP 703**。

free variable（自由變數）
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function（函式）
一連串的陳述式，它能🄵向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參🄵*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式🄵釋）
函式參數或回傳值的一個*annotation*（🄵釋）。

函式🄵釋通常被使用於型🄵提示：例如，這個函式預期會得到兩個*int* 引數，🄵會有一個*int* 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函式🄵釋的語法在 function 章節有詳細解釋。

請參🄵*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於🄵釋的最佳實踐方法，另請參🄵 annotations-howto。

__future__
future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成🄵標準的語法或語義，來編譯目前的模組。而*__future__* 模組則記🄵了 *feature*（功能）可能的值。透過 import 此模組🄵對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成🄵預設的功能：

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收）
當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能🄵檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用*gc* 模組對其進行控制。

generator（🄵生器）
一個會回傳*generator iterator*（🄵生器🄵代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能🄵生一系列的值，這些值可用於 for 🄵圈，或是以*next()* 函式，每次檢索其中的一個值。

這個術語通常用來表示一個🄵生器函式，但在某些情境中，也可能是表示🄵生器🄵代器。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

generator iterator（🄵生器🄵代器）
一個由*generator*（🄵生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，🄵記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當🄵生器🄵代器回復時，它會從停止的地方繼續執行（與那些每次呼叫時都要重新開始的函式有所不同）。

generator expression（生成器運算式）
一個會回傳生成器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了迴圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會在外層函式生成多個值：

```
>>> sum(i*i for i in range(10))         # 平方之和 0, 1, 4, ... 81
285
```

generic function（泛型函式）
一個由多個函式組成的函式，該函式會對不同的型實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來決定。

另請參*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型別）
一個能被參數化 (parameterized) 的*type*（型別）；通常是一個 容器型別，像是`list` 和`dict`。它被用於型別提示和註釋。

詳情請參泛型別、名型別、**PEP 483**、**PEP 484**、**PEP 585** 和`typing` 模組。

GIL
請參*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）
CPython 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的建型別，如`dict`）自動地避免行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標準的或是第三方的，它們被設計成在執行壓縮或雜等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

從 Python 3.13 開始可以使用 `--disable-gil` 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用 `-X gil=0` 來執行程式碼，或者設定 `PYTHON_GIL=0` 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參 **PEP 703**。

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc（雜架構的 pyc）
一個位元組碼 (bytecode) 暫存檔，它使用雜值而不是對應原始檔案的最後修改時間，來確定其有效性。請參 pyc-invalidation。

hashable（可雜的）
如果一個物件有一個雜值，該值在其生命期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那它就是一個可雜物件。比較結果相等的多個可雜物件，它們必須擁有相同的雜值。

可雜性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因這些資料結構都在其部使用了雜值。

大多數的 Python 不可變建物件都是可雜的；可變的容器（例如 list 或 dictionary）不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜的，它們本身才是可雜的。若物件是使用者自定 class 的實例，則這些物件會被預設可雜的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜值則是衍生自它們的`id()`。

IDLE
Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。*IDLE --- Python 編輯器與 shell* 是一個基本的編輯器和直譯器環境，它和 Python 的標準發行版本一起被提供。

immortal（不滅）
不滅物件 (Immortal objects) 是 **PEP 683** 引入的 CPython 實作細節。

如果一個物件是不滅的，它的參照計數永遠不會被修改，因此在直譯器運行時它永遠不會被釋放。例如，`True` 和`None` 在 CPython 中是不滅的。

Immortal objects can be identified via *sys._is_immortal()*, or via `PyUnstable_IsImmortal()` in the C API.

immutable（不可變物件）
一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要F定雜F值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）
一個位置（或*路徑項目*）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自*sys.path*，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）
一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）
一個能F尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的）
Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們F且看到它們的結果。只要F動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模組和包的非常F大的方法（請記住 help(x)）。更多互動式模式相關資訊請見 tut-interac。

interpreted（直譯的）
Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因F有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯F期，不過它們的程式通常也運行得較慢。另請參F*interactive*（互動的）。

interpreter shutdown（直譯器關閉）
當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵F部結構。它也會多次呼叫*垃圾回收器 (garbage collector)*。這能F觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，F執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因F它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的F本已經執行完成。

iterable（可F代物件）
一種能F一次回傳一個其中成員的物件。可F代物件的例子包括所有的序列型F（像是*list*、*str* 和*tuple*）和某些非序列型F，像是*dict*、*檔案物件*，以及你所定義的任何 class 物件，只要那些 class 有實作*sequence*（序列）語意的 `__iter__()` 或是 `__getitem__()` method，該物件就是可F代物件。

可F代物件可用於 `for` F圈和許多其他需要一個序列的地方 (*zip()*、*map()*...)。當一個可F代物件作F引數被傳遞給F建函式*iter()* 時，它會F該物件回傳一個F代器。此F代器適用於針對一組值進行一遍 (one pass) 運算。使用F代器時，通常不一定要呼叫*iter()* 或自行處理F代器物件。`for` 陳述式會自動地F你處理這些事，它會建立一個暫時性的未命名變數，用於在F圈期間保有該F代器。另請參F*iterator*（F代器）、*sequence*（序列）和*generator*（F生器）。

iterator（F代器）
一個表示資料流的物件。重F地呼叫F代器的*__next__()* method（或是將它傳遞給F建函式*next()*）會依序回傳資料流中的各項目。當不再有資料時，則會引發*StopIteration* 例外。此時，該F代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發*StopIteration*。F代器必須有一個*__iter__()* method，它會回傳F代器物件本身，所以每個F代器也都是可F代物件，且可以用於大多數適用其他可F代物件的場合。一個明顯的例外，是嘗試多遍F代 (multiple iteration passes) 的程式碼。一個容器物件（像是*list*）在每次你將它傳遞給*iter()* 函式或在 `for` F圈中使用它時，都會F生一個全新的F代器。使用F代器嘗試此事（多遍F代）時，只會回傳在前一遍F代中被用過的、同一個已被用盡的F代器物件，使其看起來就像一個空的容器。

在F代器型F文中可以找到更多資訊。

CPython 🄵不是始終如一地都會檢查「🄵代器有定義 *__iter__()*」這個規定。另請注意，free-threading（自由執行緒）CPython 不保證🄵代器操作的執行緒安全。

key function（鍵函式）
鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，*locale.strxfrm()* 被用來🄵生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括*min()*、*max()*、*sorted()*、*list.sort()*、*heapq.merge()*、*heapq.nsmallest()*、*heapq.nlargest()* 和*itertools.groupby()*。

有幾種方法可以建立一個鍵函式。例如，*str.lower()* method 可以作🄵不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，*operator.attrgetter()*、*operator.itemgetter()* 和*operator.methodcaller()* 🄵三個鍵函式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參🄵如何排序。

keyword argument（關鍵字引數）
請參🄵*argument*（引數）。

lambda
由單一*expression*（運算式）所組成的一個匿名行🄵函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL
Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先🄵條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競🄵條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解🄵。

lexical analyzer（詞法分析器）
tokenizer 的正式名稱；請參🄵*token*。

list（串列）
一個 Python 🄵建的*sequence*（序列）。🄵管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 *O*(1)。

list comprehension（串列綜合運算）
一種用來處理一個序列中的全部或部分元素，🄵將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會🄵生一個字串 list，其中包含 0 到 255 範圍🄵，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器）
一個能🄵載入模組的物件。它必須定義 `exec_module()` 和 `create_module()` 方法以實作*Loader* 介面。載入器通常是被*finder*（尋檢器）回傳。更多細節請參🄵：

- finders-and-loaders
- *importlib.abc.Loader*
- **PEP 302**

locale encoding（區域編碼）
在 Unix 上，它是 LC_CTYPE 區域設定的編碼。它可以用*locale.setlocale(locale.LC_CTYPE, new_locale)* 來設定。

在 Windows 上，它是 ANSI 碼頁（code page，例如 `"cp1252"`）。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作🄵區域編碼。

locale.getencoding() 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

magic method（魔術方法）
special method（特殊方法）的一個非正式同義詞。

mapping（對映）
一個容器物件，它支援任意鍵的查找，且能實作*abstract base classes*（抽象基底類🄵）中，`collections.abc.Mapping` 或`collections.abc.MutableMapping` 所指定的 method。範例包括`dict`、`collections.defaultdict`、`collections.OrderedDict` 和`collections.Counter`。

meta path finder（元路徑尋檢器）
一種經由搜尋`sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參🄵`importlib.abc.MetaPathFinder`。

metaclass（元類🄵）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類🄵）的列表。Metaclass 負責接受這三個引數，🄵建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特🄵之處在於它能🄵建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供🄵大且優雅的解🄵方案。它們已被用於記🄵屬性存取、增加執行緒安全性、追🄵物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體🄵被定義的函式。如果 method 作🄵其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成🄵它的第一個*argument*（引數）（此引數通常被稱🄵 `self`）。請參🄵*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類🄵）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參🄵 python_2.3_mro。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參🄵*package*（套件）。

module spec（模組規格）
一個命名空間，它包含用於載入模組的 import 相關資訊。它是`importlib.machinery.ModuleSpec` 的一個實例。

另請參🄵 module-specs。

MRO
請參🄵*method resolution order*（方法解析順序）。

mutable（可變物件）
可變物件可以改變它們的值，但維持它們的`id()`。另請參🄵*immutable*（不可變物件）。

named tuple（附名元組）
術語「named tuple（附名元組）」是指從 tuple 繼承的任何型🄵或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型🄵或 class 也可以具有其他的特性。

有些🄵建型🄵是 named tuple，包括由`time.localtime()` 和`os.stat()` 回傳的值。另一個例子是`sys.float_info`：

```
>>> sys.float_info[1]                   # 以索引存取
1024
>>> sys.float_info.max_exp              # 以欄位名稱存取
1024
>>> isinstance(sys.float_info, tuple)   # 屬於 tuple 型🄵
True
```

有些 named tuple 是🄵建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自`tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編

寫、可以繼承自*typing.NamedTuple* 來建立，也可以使用工廠函式 (factory function) *collections.namedtuple()* 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

namespace（命名空間）
變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及🄵建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式*builtins.open* 和*os.open()* 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出*random.seed()* 或*itertools.islice()* 明確地表示，這些函式分🄵是由*random* 和*itertools* 模組在實作。

namespace package（命名空間套件）
一個*package*（套件），它只能作🄵子套件 (subpackage) 的一個容器。命名空間套件可能🄵有實體的表示法，而且具體來🄵它們不像是一個*regular package*（正規套件），因🄵它們🄵🄵有 `__init__.py` 這個檔案。

命名空間套件允許數個可獨立安裝的套件擁有一個共同的父套件。除此之外，建議使用*regular package*。

更多資訊，請參🄵 **PEP 420** 和 reference-namespace-package。

另請參🄵*module*（模組）。

nested scope（巢狀作用域）
能🄵參照外層定義 (enclosing definition) 中的變數的能力。舉例來🄵，一個函式如果是在另一個函式中被定義，則它便能🄵參照外層函式中的變數。請注意，在預設情🄵下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最🄵層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類🄵）
一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類🄵方法）和 static method（🄵態方法）。

object（物件）
具有狀態（屬性或值）及被定義的行🄵（method）的任何資料。它也是任何*new-style class*（新式類🄵）的最終 base class（基底類🄵）。

optimized scope（最佳化作用域）
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞🄵的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參🄵*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能🄵接受的一個*argument*（引數），或在某些情🄵下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作🄵關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 `*` 來定義的，例如以下的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 `**` 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以🄵一些選擇性的引數指定預設值。

另請參🄵術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差🄵、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目）
在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）
被`sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參🄵`importlib.abc.PathEntryFinder`。

path entry hook（路徑項目🄵）
在`sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）
預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）
一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的`str` 或`bytes` 物件，或是一個實作`os.PathLike` 協定的物件。透過呼叫`os.fspath()` 函式，一個支援`os.PathLike` 協定的物件可以被轉🄵🄵`str` 或`bytes` 檔案系統路徑；而`os.fsdecode()` 及`os.fsencode()` 則分🄵可用於確保`str` 及`bytes` 的結果。由 **PEP 519** 引入。

PEP
Python Enhancement Proposal（Python 增🄵提案）。PEP 是一份設計🄵明文件，它能🄵 Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成🄵重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計🄵策的記🄵，這些過程的主要機制。PEP 的作者要負責在社群🄵建立共識🄵記🄵反對意見。

請參🄵 **PEP 1**。

portion（部分）
在單一目🄵中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）
請參🄵*argument*（引數）。

provisional API（暫行 API）
暫行 API 是指，從標準函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示🄵暫行的，理論上🄵不會有重大的變更，但如果核心開發人員認

🄵有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更🄵不會無端地🄵生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視🄵「最後的解🄵方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解🄵方案。

這個過程使得標準函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參🄵 **PEP 411** 了解更多細節。

provisional package（暫行套件）
請參🄵*provisional API*（暫行 API）。

Python 3000
Python 3.x 系列版本的🄵稱（很久以前創造的，當時第 3 版的發布是在🄵遠的未來。）也可以縮寫🄵「Py3k」。

Pythonic（Python 風格的）
一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可🄵代物件的所有元素進行🄵圈。許多其他語言🄵🄵有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```
for piece in food:
    print(piece)
```

qualified name（限定名稱）
一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，*完全限定名稱 (fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count（參照計數）
對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「不滅的 (immortal)」🄵擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它🄵是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 `sys.getrefcount()` 函式來回傳一個特定物件的參照計數。

在*CPython* 中，參照計數不被視🄵穩定或明確定義的值；對物件的參照數量，以及該數量如何受到 Python 程式碼的影響，在不同版本之間可能會有所不同。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目🄵。

另請參🄵*namespace package*（命名空間套件）。

REPL
「read-eval-print 🄵圈 (read–eval–print loop)」的縮寫，是互動式直譯器 shell 的另一個名稱。

__slots__
在 class 🄵部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情🄵。

sequence（序列）
一個*iterable*（可🄵代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，🄵定義了一個 `__len__()` method 來回傳該序列的長度。一些🄵建序列型🄵包括`list`、`str`、`tuple` 和`bytes`。請注意，雖然`dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視🄵對映 (mapping) 而不是序列，因🄵其查找方式是使用任意的*hashable* 鍵，而不是整數。

抽象基底類🄵 (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，🄵不僅止於 `__getitem__()` 和 `__len__()`，還增加了`count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型🄵，可以使用`register()` 被明確地🄵🄵。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）
一種緊密的方法，用來處理一個可🄵代物件中的全部或部分元素，🄵將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會🄵生一個字串 set：`{'r', 'd'}`。請參🄵 comprehensions。

single dispatch（單一調度）
generic function（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型🄵。

slice（切片）
一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) `[]`，若要給出多個數字，則在數字之間使用冒號，例如 `variable_name[1:3:5]`。在括號（下標）符號的🄵部，會使用`slice` 物件。

soft deprecated（軟性🄵用）
被軟性🄵用的 API 代表不應再用於新程式碼中，但在現有程式碼中繼續使用它仍會是安全的。API 仍會以文件記🄵🄵會被測試，但不會被繼續改進。

與正常🄵用不同，軟性🄵用🄵有🄵除 API 的規劃，也不會發出警告。

請參🄵 PEP 387：軟性🄵用。

special method（特殊方法）
一種會被 Python 自動呼叫的 method，用於對某種型🄵執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底🄵。Special method 在 specialnames 中有詳細🄵明。

標准函式庫
包含套件、模組和擴充模組的集合，它們是作🄵官方 Python 直譯器套件的一部分來發行。該集合的成員可能會因平台、可用的系統函式庫或其他條件而有所不同。相關文件可以在 *Python 標準函式庫 (Standard Library)* 中找到。

請參🄵`sys.stdlib_module_names` 以取得所有可能的標准函式庫模組名稱的列表。

statement（陳述式）
陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

static type checker（🄵態型🄵檢查器）
會讀取 Python 程式碼🄵分析的外部工具，能🄵找出錯誤，像是使用了不正確的型🄵。另請參🄵型🄵提示 *(type hints)* 以及`typing` 模組。

stdlib（標准函式庫）
standard library 的縮寫。

strong reference（🄵參照）
在 Python 的 C API 中，🄵參照是對物件的參照，該物件🄵持有該參照的程式碼所擁有。建立參照時透過呼叫 `Py_INCREF()` 來獲得🄵參照、🄵除參照時透過 `Py_DECREF()` 釋放🄵參照。

`Py_NewRef()` 函式可用於建立一個對物件的🄵參照。通常，在退出🄵參照的作用域之前，必須在該🄵參照上呼叫 `Py_DECREF()` 函式，以避免🄵漏一個參照。

另請參🄵*borrowed reference*（借用參照）。

t-string（t 字串）
t-strings（t 字串）
以 `t` 或 `T` 🄵前綴的字串文本通常被稱🄵「t 字串」，它是模板化的字串文本的縮寫。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 編碼位置 (code point) 的序列（範圍在 `U+0000` -- `U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化🄵一個位元組序列。

將一個字串序列化🄵位元組序列，稱🄵「編碼」，而從位元組序列重新建立該字串則稱🄵「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (*codecs*)，它們被統稱🄵「文字編碼」。

text file（文字檔案）
一個能🄵讀取和寫入*`str`* 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) 🄵會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開🄵的檔案、*`sys.stdin`*、*`sys.stdout`* 以及*`io.StringIO`* 的實例。

另請參🄵*binary file*（二進位檔案），它是一個能🄵讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python's C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token
原始碼的小單位，由 詞法分析器（也稱🄵 *tokenizer*）🄵生。名稱、數字、字串、運算子、🄵行符號等都以 token 表示。

`tokenize` 模組公開了 Python 的詞法分析器。*`token`* 模組包含各種 token 類型的資訊。

triple-quoted string（三引號🄵字串）
由三個雙引號 (") 或單引號 (') 的作🄵邊界的一個字串。雖然它們🄵🄵有提供🄵於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳🄵 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫🄵明字串時特🄵有用。

type（型🄵）
一個 Python 物件的型🄵🄵定了它是什🄵類型的物件；每個物件都有一個型🄵。一個物件的型🄵可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型🄵🄵名）
一個型🄵的同義詞，透過將型🄵指定給一個識🄵符 (identifier) 來建立。

型🄵🄵名對於簡化型🄵提示 *(type hint)* 很有用。例如：

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以寫成這樣，更具有可讀性：

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

請參🄵*typing* 和 **PEP 484**，有此功能的描述。

type hint（型🄵提示）
一種*annotation*（🄵釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型🄵。

型🄵提示是選擇性的，而不是被 Python 🄵制的，但它們對🄵態型🄵檢查器 *(static type checkers)*很有用，🄵能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型🄵提示，都可以使用*typing.get_type_hints()* 來存取。

請參🄵*typing* 和 **PEP 484**，有此功能的描述。

universal newlines（通用🄵行字元）
一種解譯文字流 (text stream) 的方式，會將以下所有的情🄵識🄵🄵一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參🄵 **PEP 278** 和 **PEP 3116**，以及用於*bytes.splitlines()* 的附加用途。

variable annotation（變數🄵釋）
一個變數或 class 屬性的*annotation*（🄵釋）。

🄵釋變數或 class 屬性時，賦值是選擇性的：

```
class C:
    field: 'annotation'
```

變數🄵釋通常用於型🄵提示 *(type hint)*：例如，這個變數預期會取得*int*（整數）值：

```
count: int = 0
```

變數🄵釋的語法在 annassign 章節有詳細的解釋。

請參🄵*function annotation*（函式🄵釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於🄵釋的最佳實踐方法，另請參🄵 annotations-howto。

virtual environment（🄵擬環境）
一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發🄵套件，而不會對同一個系統上運行的其他 Python 應用程式的行🄵🄵生干擾。

另請參🄵*venv*。

virtual machine（🄵擬機器）
一部完全由軟體所定義的電腦 (computer)。Python 的🄵擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

walrus operator（海象運算子）
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python（Python 之🄵）
Python 設計原則與哲學的列表，其🄵容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這份🄵明文件

Python 🄵明文件是透過使用 Sphinx（一個原🄵 Python 而生的文件🄵生器、目前是以獨立專案形式來維護）將使用 reStructuredText 撰寫的原始檔轉🄵而成。

如同 Python 自身，透過自願者的努力下🄵出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，🄵含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者；
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾🄵 Python 這門語言、Python 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl）的 Guido van Rossum 於 1990 年代早期所創造，目的是作🄵一種稱🄵 ABC 語言的後繼者。🄵管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us）繼續他在 Python 的工作，🄵在那🄵發🄵了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com 🄵成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成🄵 Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專🄵擁有 Python 相關的智慧財🄵權而創立的非營利組織。Zope Corporation 過去是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參🄵 https://opensource.org）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差🄵。

發🄵版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備🄵

(1) GPL 相容🄵不表示我們是在 GPL 下發🄵 Python。不像 GPL，所有的 Python 授權都可以讓你發🄵修改後的版本，但不一定要使你的變更成🄵開源。GPL 相容的授權使得 Python 可以結合其他

在 GPL 下發🄵的軟體一起使用；但其它的授權則不行。

(2) 根據 Richard Stallman 的🄵法，1.6.1 不是 GPL 相容的，因🄵其授權有一個法律選擇條款。然而根據 CNRI 的🄵法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發🄵成🄵可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和🄵明文件的授權是基於 Python 軟體基金會授權第二版 (Python Software Foundation License Version 2)。

從 Python 3.8.6 開始，🄵明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權第二版以及*Zero-Clause BSD* 授權。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參🄵被收🄵軟體的授權與致謝。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

(繼續下一頁)

(繼續上一頁)

```
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(繼續下一頁)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 被收🄵軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發🄵版本中所收🄴的第三方軟體。

C.3.1 Mersenne Twister

random 模組底下的 _random C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載🄵容🄴基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

(繼續下一頁)

(繼續上一頁)

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）🄵，於不同的原始檔案中被編碼：

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

`test.support.asynchat` 和 `test.support.asyncore` 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
```

(繼續下一頁)

(繼續上一頁)

```
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

http.cookies 模組包含以下聲明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

                All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追🄵

trace 模組包含以下聲明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
```

(繼續下一頁)

(繼續上一頁)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 編解碼器包含以下聲明：

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(繼續下一頁)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明：

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明：

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

`Python/pyhash.c` 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

`Python/dtoa.c` 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉🄵。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則*hashlib*、*posix*、*ssl* 模組會使用它來提升效能。此外，因🄵 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收🄵 OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2：

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/
```

(繼續下一頁)

(繼續上一頁)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

(繼續下一頁)

(繼續上一頁)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(繼續下一頁)

(繼續上一頁)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

除非在建置*pyexpat* 擴充時設定🄵 `--with-system-expat`，否則該擴充會用一個🄵含 expat 原始碼的副本來建置:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper
```

(繼續下一頁)

(繼續上一頁)

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 _ctypes 模組底下 _ctypes 擴充程式時設定🄵 `--with-system-libffi`，否則該擴充會用一個🄴含 libffi 原始碼的副本來建置：

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置*zlib* 擴充，則該擴充會用一個🄴含 zlib 原始碼的副本來建置：

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
```

(繼續下一頁)

(繼續上一頁)

```
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的雜🄵表 (hash table) 實作，是以 cfuhash 專案🄵基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

除非在建置*decimal* 模組底下 `_decimal` C 擴充程式時設定🄵 `--with-system-libmpdec`，否則該模組會用一個🄵含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

(繼續下一頁)

(繼續上一頁)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

test 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發🄵：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT 授權：

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

asyncio 模組的部分🄴容是從 uvloop 0.16 中收🄴過來，其基於 MIT 授權來發🄵：

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
```

(繼續下一頁)

(繼續上一頁)

```
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

版權宣告

Python 和這份說明文件的版權:

Copyright © 2001 Python 軟體基金會。保留所有權利。

Copyright © 2000 BeOpen.com 保留所有權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留所有權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留所有權利。

完整的授權條款資訊請參見沿革與授權。

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. The third edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

[C99] ISO/IEC 9899:1999. "Programming languages -- C." 公開草案可在以下網址取得 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf。

Python 模組索引

_

__future__, 2010
__main__, 1958
_thread, 1071
_tkinter, 1582

a

abc, 1994
aifc, 2219
annotationlib, 2035
argparse, 855
array, 293
ast, 2093
asynchat, 2219
asyncio, 1075
asyncore, 2219
atexit, 1999
audioop, 2220

b

base64, 1352
bdb, 1859
binascii, 1356
bisect, 290
builtins, 1957
bz2, 581

c

calendar, 255
cgi, 2220
cgitb, 2220
chunk, 2220
cmath, 349
cmd, 959
code, 2047
codecs, 194
codeop, 2049
collections, 263
collections.abc, 280
colorsys, 1560
compileall, 2147
compression.zstd, 563
concurrent.futures, 1031
concurrent.interpreters, 1039
configparser, 626
contextlib, 1980
contextvars, 1067
copy, 309
copyreg, 520
cProfile, 1880
crypt, 2220
csv, 619
ctypes, 817
curses (*Unix*), 928
curses.ascii, 954
curses.panel, 958
curses.textpad, 953

d

dataclasses, 1969
datetime, 213
dbm, 525
dbm.dumb, 530
dbm.gnu (*Unix*), 528
dbm.ndbm (*Unix*), 529
dbm.sqlite3 (*All*), 527
decimal, 353
difflib, 161
dis, 2151
distutils, 2220
doctest, 1724

e

email, 1263
email.charset, 1313
email.contentmanager, 1292
email.encoders, 1315
email.errors, 1285
email.generator, 1275
email.header, 1311
email.headerregistry, 1286
email.iterators, 1318
email.message, 1264
email.mime, 1308
email.mime.application, 1309
email.mime.audio, 1309
email.mime.base, 1308

email.mime.image, 1310
email.mime.message, 1310
email.mime.multipart, 1309
email.mime.nonmultipart, 1309
email.mime.text, 1310
email.parser, 1272
email.policy, 1278
email.utils, 1316
encodings, 209
encodings.idna, 210
encodings.mbcs, 211
encodings.utf_8_sig, 211
ensurepip, 1903
enum, 319
errno, 808

f

faulthandler, 1864
fcntl (*Unix*), 2203
filecmp, 481
fileinput, 926
fnmatch, 490
fractions, 381
ftplib, 1474
functools, 425

g

gc, 2012
getopt, 2215
getpass, 925
gettext, 1561
glob, 488
graphlib, 334
grp (*Unix*), 2198
gzip, 577

h

hashlib, 651
heapq, 286
hmac, 662
html, 1359
html.entities, 1364
html.parser, 1360
http, 1463
http.client, 1466
http.cookiejar, 1522
http.cookies, 1518
http.server, 1511

i

idlelib, 1628
imaplib, 1483
imghdr, 2221
imp, 2221
importlib, 2059
importlib.abc, 2062
importlib.machinery, 2068
importlib.metadata, 2083
importlib.resources, 2079
importlib.resources.abc, 2082
importlib.util, 2073
inspect, 2016
io, 734
ipaddress, 1543
itertools, 409

j

json, 1319
json.tool, 1328

k

keyword, 2139

l

linecache, 492
locale, 1569
logging, 759
logging.config, 778
logging.handlers, 790
lzma, 585

m

mailbox, 1329
mailcap, 2221
marshal, 524
math, 340
mimetypes, 1348
mmap, 1256
modulefinder, 2055
msilib, 2221
msvcrt (*Windows*), 2177
multiprocessing, 982
multiprocessing.connection, 1013
multiprocessing.dummy, 1017
multiprocessing.managers, 1004
multiprocessing.pool, 1010
multiprocessing.shared_memory, 1026
multiprocessing.sharedctypes, 1002

n

netrc, 646
nis, 2221
nntplib, 2221
numbers, 337

o

operator, 435
optparse, 898
os, 667
os.path, 468
ossaudiodev, 2222

p

pathlib, 443
pathlib.types, 468
pdb, 1868

pickle, 505
pickletools, 2174
pipes, 2222
pkgutil, 2053
platform, 804
plistlib, 647
poplib, 1480
posix (*Unix*), 2196
pprint, 311
profile, 1880
pstats, 1881
pty (*Unix*), 2201
pwd (*Unix*), 2197
py_compile, 2145
pyclbr, 2144
pydoc, 1720

q

queue, 1064
quopri, 1358

r

random, 384
re, 139
readline (*Unix*), 180
reprlib, 316
resource (*Unix*), 2206
rlcompleter, 184
runpy, 2057

s

sched, 1063
secrets, 663
select, 1237
selectors, 1243
shelve, 521
shlex, 2191
shutil, 492
signal, 1247
site, 2043
sitecustomize, 2044
smtpd, 2222
smtplib, 1490
sndhdr, 2222
socket, 1173
socketserver, 1503
spwd, 2222
sqlite3, 531
ssl, 1202
stat, 475
statistics, 393
string, 123
string.templatelib, 134
stringprep, 179
struct, 187
subprocess, 1044
sunau, 2222
symtable, 2130
sys, 1919
sys.monitoring, 1946
sysconfig, 1951
syslog (*Unix*), 2210

t

tabnanny, 2143
tarfile, 602
telnetlib, 2223
tempfile, 483
termios (*Unix*), 2199
test, 1834
test.regrtest, 1836
test.support, 1836
test.support.bytecode_helper, 1847
test.support.import_helper, 1851
test.support.os_helper, 1849
test.support.script_helper, 1846
test.support.socket_helper, 1846
test.support.threading_helper, 1848
test.support.warnings_helper, 1852
textwrap, 173
threading, 965
time, 748
timeit, 1886
tkinter, 1579
tkinter.colorchooser (*Tk*), 1591
tkinter.commondialog (*Tk*), 1595
tkinter.dnd (*Tk*), 1598
tkinter.filedialog (*Tk*), 1593
tkinter.font (*Tk*), 1591
tkinter.messagebox (*Tk*), 1595
tkinter.scrolledtext (*Tk*), 1597
tkinter.simpledialog (*Tk*), 1592
tkinter.ttk, 1599
token, 2134
tokenize, 2139
tomllib, 644
trace, 1890
traceback, 2000
tracemalloc, 1893
tty (*Unix*), 2200
turtle, 1628
turtledemo, 1663
types, 303
typing, 1667

u

unicodedata, 176
unittest, 1747
unittest.mock, 1777
urllib, 1435
urllib.error, 1461
urllib.parse, 1452
urllib.request, 1435
urllib.response, 1452
urllib.robotparser, 1462
usercustomize, 2044

uu, 2223
uuid, 1497

V

venv, 1905

W

warnings, 1962
wave, 1557
weakref, 296
webbrowser, 1423
winreg (*Windows*), 2180
winsound (*Windows*), 2188
wsgiref, 1426
wsgiref.handlers, 1430
wsgiref.headers, 1428
wsgiref.simple_server, 1428
wsgiref.types, 1433
wsgiref.util, 1426
wsgiref.validate, 1429

X

xdrlib, 2223
xml, 1364
xml.dom, 1384
xml.dom.minidom, 1394
xml.dom.pulldom, 1398
xml.etree.ElementInclude, 1376
xml.etree.ElementTree, 1365
xml.parsers.expat, 1411
xml.parsers.expat.errors, 1418
xml.parsers.expat.model, 1418
xml.sax, 1400
xml.sax.handler, 1401
xml.sax.saxutils, 1406
xml.sax.xmlreader, 1407
xmlrpc, 1530
xmlrpc.client, 1530
xmlrpc.server, 1537

Z

zipapp, 1914
zipfile, 591
zipimport, 2051
zlib, 574
zoneinfo, 251

索引

非依字母順序

??
    於正規表示式中, 140
..
    於 pathnames (路徑名稱) 中, 732
..., **2227**
    ellipsis literal, 1670, 1672, 1682
    ellipsis literal (🄵節號字面值), 36, 104
    interpreter prompt (直譯器提示), 1728, 1939
    placeholder (🄵位符號), 176, 312, 317
    🄵節號字面值, 2227
    於 doctests 中, 1731
! (*pdb command*), 1876
! formatted string literals (! 格式化字串常數), 63
! f-string (! f 字串), 63
? (問號)
    replacement character (替代字元), 197
    於 argparse 模組中, 867
    於 AST 文法中, 2096
    於 command interpreter (命令直譯器) 中, 960
    於 glob 風格的萬用字元中, 488, 490
    於 SQL 陳述式中, 548
    於 struct format strings (結構格式字串), 190
    於正規表示式中, 140
- (🄵號)
    binary operator (二元運算子), 38
    unary operator (一元運算子), 38
    於 doctests 中, 1732
    於 glob 風格的萬用字元中, 488, 490
    於 printf 風格格式化, 65, 81
    於字串格式化, 127
    於正規表示式中, 141
! (驚嘆號)
    於 command interpreter (命令直譯器) 中, 960
    於 curses 模組中, 958
    於 glob 風格的萬用字元中, 488, 490
    於 struct format strings (結構格式字串), 188
    於字串格式化, 125
    於格式化字串常數中, 63
. (點)
    於 glob 風格的萬用字元中, 488
    於 pathnames (路徑名稱) 中, 732
    於 printf 風格格式化, 65, 80
    於字串格式化, 125
    於正規表示式中, 140
(井字號)
    comment (🄵解), 2043
    於 doctests 中, 1732
    於 printf 風格格式化, 65, 81
    於字串格式化, 127
    於正規表示式中, 148
$ (金錢符號)
    environment variables expansion (環境變數展開), 470
    interpolation in configuration files (設定檔中的插值), 631
    於 template strings (模板字串), 132
    於正規表示式中, 140
% (百分號)
    datetime format (日期時間格式), 246, 752, 754
    environment variables expansion (Windows) (環境變數展開 (Windows)), 2182
    environment variables expansion (Windows) (環境變數展開 (Windows)), 470
    interpolation in configuration files (設定檔中的插值), 631
    operator (運算子), 38
    printf 風格格式化, 64, 80
& (和號)
    operator (運算子), 40
(?
    於正規表示式中, 142
(?!
    於正規表示式中, 143
(?#
    於正規表示式中, 143
(?(
    於正規表示式中, 144

() (圓括號)
於 printf 風格格式化, 65, 80
於正規表示式中, 142
(?:
於正規表示式中, 142
(?<!
於正規表示式中, 144
(?<=
於正規表示式中, 144
(?=
於正規表示式中, 143
(?P<
於正規表示式中, 143
(?P=
於正規表示式中, 143
*?
於正規表示式中, 140
* (星號)
operator (運算子), 38
於 argparse 模組中, 867
於 AST 文法中, 2096
於 glob 風格的萬用字元中, 488, 490
於 printf 風格格式化, 65, 80
於正規表示式中, 140
**
operator (運算子), 38
於 glob 風格的萬用字元中, 488
*+
於正規表示式中, 140
+?
於正規表示式中, 140
?+
於正規表示式中, 140
+ (加號)
binary operator (二元運算子), 38
unary operator (一元運算子), 38
於 argparse 模組中, 867
於 doctests 中, 1732
於 printf 風格格式化, 65, 81
於字串格式化, 127
於正規表示式中, 140
++
於正規表示式中, 140
, (逗號)
於字串格式化, 127
-
idle 命令列選項, 1624
python--m-py_compile 命令列選項, 2147
/ (斜🄵)
operator (運算子), 38
於 pathnames (路徑名稱) 中, 732
//
operator (運算子), 38
2-digit years (2 位數年份), 748
: (冒號)
path separator (POSIX) (路徑分隔器 (POSIX)), 732
於 SQL 陳述式中, 548
於字串格式化, 125
於格式化字串常數中, 63
; (分號), 732
< (小於)
operator (運算子), 38
於 struct format strings (結構格式字串), 188
於字串格式化, 127
<<
operator (運算子), 40
<=
operator (運算子), 38
<BLANKLINE>, 1731
<file>
python--m-py_compile 命令列選項, 2147
!=
operator (運算子), 38
= (等號)
for help in debugging using string literals, 63
於 struct format strings (結構格式字串), 188
於字串格式化, 127
==
operator (運算子), 38
> (大於)
operator (運算子), 38
於 struct format strings (結構格式字串), 188
於字串格式化, 127
>=
operator (運算子), 38
>>
operator (運算子), 40
>>>, **2227**
interpreter prompt (直譯器提示), 1728, 1939
@ (在)
於 struct format strings (結構格式字串), 188
[] (方括號)
於 glob 風格的萬用字元中, 488, 490
於字串格式化, 125
於正規表示式中, 141
\ (反斜🄵)
escape sequence (跳🄵序列), 197
in pathnames (Windows) (在路徑名稱中 (Windows)), 732
於正規表示式中, 141, 144
\\
於正規表示式中, 146
\A
於正規表示式中, 144
\a
於正規表示式中, 146
\B
於正規表示式中, 145
\b

    於正規表示式中, 144, 146
\D
    於正規表示式中, 145
\d
    於正規表示式中, 145
\f
    於正規表示式中, 146
\g
    於正規表示式中, 150
\N
    escape sequence（跳🄵序列）, 197
    於正規表示式中, 146
\n
    於正規表示式中, 146
\r
    於正規表示式中, 146
\S
    於正規表示式中, 145
\s
    於正規表示式中, 145
\t
    於正規表示式中, 146
\U
    escape sequence（跳🄵序列）, 197
    於正規表示式中, 146
\u
    escape sequence（跳🄵序列）, 197
    於正規表示式中, 146
\v
    於正規表示式中, 146
\W
    於正規表示式中, 145
\w
    於正規表示式中, 145
\x
    escape sequence（跳🄵序列）, 197
    於正規表示式中, 146
\Z
    於正規表示式中, 145
\z
    於正規表示式中, 145
^（插入符號）
    marker（標記）, 1730, 2001
    operator（運算子）, 40
    於 curses 模組中, 958
    於字串格式化, 127
    於正規表示式中, 140, 141
_（底🄵）
    gettext, 1562
    於字串格式化, 127
__abs__() (於 operator 模組中), 436
__add__() (於 operator 模組中), 436
__and__() (enum.Flag 的方法), 328
__and__() (於 operator 模組中), 436
__args__ (genericalias 的屬性), 100
__bound__ (typing.TypeVar 的屬性), 1692
__breakpointhook__ (於 sys 模組中), 1923
__bytes__() (email.message.EmailMessage 的方法), 1265
__bytes__() (email.message.Message 的方法), 1302
__call__() (argparse.Action 的方法), 873
__call__() (email.headerregistry.HeaderRegistry 的方法), 1290
__call__() (enum.EnumType 的方法), 321
__call__() (於 operator 模組中), 438
__call__() (weakref.finalize 的方法), 299
__callback__ (weakref.ref 的屬性), 297
__cause__ (BaseException 的屬性), 109
__cause__ (traceback.TracebackException 的屬性), 2003
__cause__（例外屬性）, 109
__ceil__() (fractions.Fraction 的方法), 383
__class__ (unittest.mock.Mock 的屬性), 1786
__code__（函式物件屬性）, 104
__concat__() (於 operator 模組中), 437
__constraints__ (typing.TypeVar 的屬性), 1692
__contains__() (email.message.EmailMessage 的方法), 1266
__contains__() (email.message.Message 的方法), 1303
__contains__() (enum.EnumType 的方法), 322
__contains__() (enum.Flag 的方法), 327
__contains__() (mailbox.Mailbox 的方法), 1331
__contains__() (於 operator 模組中), 437
__context__ (BaseException 的屬性), 109
__context__ (traceback.TracebackException 的屬性), 2003
__context__（例外屬性）, 109
__contravariant__ (typing.TypeVar 的屬性), 1692
__copy__()（🄵🄵協定）, 310
__covariant__ (typing.TypeVar 的屬性), 1692
__debug__ (🄵建變數), 36
__deepcopy__()（🄵🄵協定）, 310
__default__ (typing.ParamSpec 的屬性), 1696
__default__ (typing.TypeVar 的屬性), 1692
__default__ (typing.TypeVarTuple 的屬性), 1694
__del__() (io.IOBase 的方法), 739
__delitem__() (email.message.EmailMessage 的方法), 1266
__delitem__() (email.message.Message 的方法), 1304
__delitem__() (mailbox.Mailbox 的方法), 1330
__delitem__() (mailbox.MH 的方法), 1337
__delitem__() (於 operator 模組中), 438
__dir__() (enum.Enum 的方法), 323
__dir__() (enum.EnumType 的方法), 322
__dir__() (unittest.mock.Mock 的方法), 1783
__displayhook__ (於 sys 模組中), 1923
__doc__ (definition 的屬性), 105
__enter__() (contextmanager 的方法), 96
__enter__() (winreg.PyHKEY 的方法), 2188
__eq__() (email.charset.Charset 的方法), 1315
__eq__() (email.header.Header 的方法), 1312
__eq__() (memoryview 的方法), 83
__eq__() (於 operator 模組中), 435

__eq__() (實例方法), 38
__excepthook__ (於 sys 模組中), 1923
__excepthook__ (於 threading 模組中), 967
__exit__() (contextmanager 的方法), 97
__exit__() (winreg.PyHKEY 的方法), 2188
__floor__() (fractions.Fraction 的方法), 383
__floordiv__() (於 operator 模組中), 436
__format__, 16
__format__() (datetime.date 的方法), 223
__format__() (datetime.datetime 的方法), 233
__format__() (datetime.time 的方法), 238
__format__() (enum.Enum 的方法), 325
__format__() (fractions.Fraction 的方法), 383
__format__() (ipaddress.IPv4Address 的方法), 1546
__format__() (ipaddress.IPv6Address 的方法), 1548
__forward_arg__ (annotationlib.ForwardRef 的屬性), 2037
__fspath__() (os.PathLike 的方法), 671
__future__, **2233**
    module, 2010
__ge__() (於 operator 模組中), 435
__ge__() (實例方法), 38
__getitem__() (email.headerregistry.HeaderRegistry 的方法), 1290
__getitem__() (email.message.EmailMessage 的方法), 1266
__getitem__() (email.message.Message 的方法), 1303
__getitem__() (enum.EnumType 的方法), 322
__getitem__() (mailbox.Mailbox 的方法), 1331
__getitem__() (re.Match 的方法), 154
__getitem__() (於 operator 模組中), 438
__getnewargs__() (object 的方法), 511
__getnewargs_ex__() (object 的方法), 511
__getstate__() (object 的方法), 511
__getstate__() (copy 協定), 515
__gt__() (於 operator 模組中), 435
__gt__() (實例方法), 38
__iadd__() (於 operator 模組中), 441
__iand__() (於 operator 模組中), 441
__iconcat__() (於 operator 模組中), 441
__ifloordiv__() (於 operator 模組中), 441
__ilshift__() (於 operator 模組中), 441
__imatmul__() (於 operator 模組中), 441
__imod__() (於 operator 模組中), 441
__import__()
    built-in function, 32
__import__() (於 importlib 模組中), 2060
__imul__() (於 operator 模組中), 441
__index__() (於 operator 模組中), 436
__infer_variance__ (typing.TypeVar 的屬性), 1692
__init__() (asyncio.Future 的方法), 1163
__init__() (asyncio.Task 的方法), 1163
__init__() (difflib.HtmlDiff 的方法), 161
__init__() (enum.Enum 的方法), 324
__init__() (logging.Handler 的方法), 766
__init_subclass__() (enum.Enum 的方法), 324
__interactivehook__ (於 sys 模組中), 1934
__inv__() (於 operator 模組中), 436
__invert__() (於 operator 模組中), 436
__ior__() (於 operator 模組中), 441
__ipow__() (於 operator 模組中), 441
__irshift__() (於 operator 模組中), 441
__isub__() (於 operator 模組中), 442
__iter__() (container 的方法), 45
__iter__() (enum.EnumType 的方法), 322
__iter__() (iterator 的方法), 45
__iter__() (mailbox.Mailbox 的方法), 1330
__iter__() (unittest.TestSuite 的方法), 1767
__itruediv__() (於 operator 模組中), 442
__ixor__() (於 operator 模組中), 442
__le__() (於 operator 模組中), 435
__le__() (實例方法), 38
__len__() (email.message.EmailMessage 的方法), 1266
__len__() (email.message.Message 的方法), 1303
__len__() (enum.EnumType 的方法), 322
__len__() (mailbox.Mailbox 的方法), 1331
__lshift__() (於 operator 模組中), 437
__lt__() (於 operator 模組中), 435
__lt__() (實例方法), 38
__main__
    module, 1958
    module (模組), 2057, 2058
__matmul__() (於 operator 模組中), 437
__members__ (enum.EnumType 的屬性), 322
__missing__(), 93
__missing__() (collections.defaultdict 的方法), 272
__mod__() (於 operator 模組中), 437
__module__ (definition 的屬性), 105
__module__ (typing.NewType 的屬性), 1700
__module__ (typing.TypeAliasType 的屬性), 1697
__mul__() (於 operator 模組中), 437
__mutable_keys__ (typing.TypedDict 的屬性), 1704
__name__ (definition 的屬性), 105
__name__ (property 的屬性), 26
__name__ (typing.NewType 的屬性), 1700
__name__ (typing.ParamSpec 的屬性), 1696
__name__ (typing.TypeAliasType 的屬性), 1697
__name__ (typing.TypeVar 的屬性), 1692
__name__ (typing.TypeVarTuple 的屬性), 1694
__ne__() (email.charset.Charset 的方法), 1315
__ne__() (email.header.Header 的方法), 1312
__ne__() (於 operator 模組中), 435
__ne__() (實例方法), 38
__neg__() (於 operator 模組中), 437
__new__() (enum.Enum 的方法), 324
__new__() (string.templatelib.Interpolation 的方法), 138
__new__() (string.templatelib.Template 的方法), 136
__next__() (csv.csvreader 的方法), 624
__next__() (iterator 的方法), 45
__not__() (於 operator 模組中), 436
__notes__ (BaseException 的屬性), 110

__notes__ (*traceback.TracebackException* 的屬性), 2003
__optional_keys__ (*typing.TypedDict* 的屬性), 1704
__or__() (*enum.Flag* 的方法), 328
__or__() (於 *operator* 模組中), 437
__origin__ (*genericalias* 的屬性), 100
__parameters__ (*genericalias* 的屬性), 100
__pointer_type__ (*ctypes._CData* 的屬性), 846
__pos__() (於 *operator* 模組中), 437
__post_init__() (於 *dataclasses* 模組中), 1976
__pow__() (於 *operator* 模組中), 437
__qualname__ (*definition* 的屬性), 105
__readonly_keys__ (*typing.TypedDict* 的屬性), 1704
__reduce__() (*object* 的方法), 512
__reduce_ex__() (*object* 的方法), 513
__replace__() （取代協定）, 310
__repr__() (*enum.Enum* 的方法), 325
__repr__() (*multiprocessing.managers.BaseProxy* 的方法), 1010
__repr__() (*netrc.netrc* 的方法), 647
__required_keys__ (*typing.TypedDict* 的屬性), 1704
__reversed__() (*enum.EnumType* 的方法), 322
__round__() (*fractions.Fraction* 的方法), 383
__rshift__() (於 *operator* 模組中), 437
__setitem__() (*email.message.EmailMessage* 的方法), 1266
__setitem__() (*email.message.Message* 的方法), 1303
__setitem__() (*mailbox.Mailbox* 的方法), 1330
__setitem__() (*mailbox.Maildir* 的方法), 1334
__setitem__() (於 *operator* 模組中), 438
__setstate__() (*object* 的方法), 512
__setstate__() (copy 協定）, 515
__slots__, **2241**
__stderr__ (於 *sys* 模組中), 1944
__stdin__ (於 *sys* 模組中), 1944
__stdout__ (於 *sys* 模組中), 1944
__str__() (*datetime.date* 的方法), 223
__str__() (*datetime.datetime* 的方法), 233
__str__() (*datetime.time* 的方法), 238
__str__() (*email.charset.Charset* 的方法), 1314
__str__() (*email.header.Header* 的方法), 1312
__str__() (*email.headerregistry.Address* 的方法), 1291
__str__() (*email.headerregistry.Group* 的方法), 1291
__str__() (*email.message.EmailMessage* 的方法), 1265
__str__() (*email.message.Message* 的方法), 1301
__str__() (*enum.Enum* 的方法), 325
__str__() (*multiprocessing.managers.BaseProxy* 的方法), 1010
__sub__() (於 *operator* 模組中), 437
__subclasshook__() (*abc.ABCMeta* 的方法), 1995
__supertype__ (*typing.NewType* 的屬性), 1700
__suppress_context__ (*BaseException* 的屬性), 109
__suppress_context__ (*traceback.TracebackException* 的屬性), 2003
__suppress_context__（例外屬性）, 109
__test__ (*module* 的屬性), 1727
__total__ (*typing.TypedDict* 的屬性), 1703
__traceback__ (*BaseException* 的屬性), 110
__truediv__() (*importlib.abc.Traversable* 的方法), 2067
__truediv__() (*importlib.resources.abc.Traversable* 的方法), 2083
__truediv__() (於 *operator* 模組中), 437
__type_params__ (*definition* 的屬性), 105
__type_params__ (*typing.TypeAliasType* 的屬性), 1697
__unpacked__ (*genericalias* 的屬性), 100
__unraisablehook__ (於 *sys* 模組中), 1923
__value__ (*typing.TypeAliasType* 的屬性), 1697
__version__ (於 *curses* 模組中), 942
__xor__() (*enum.Flag* 的方法), 328
__xor__() (於 *operator* 模組中), 437
_add_alias_() (*enum.Enum* 的方法), 325
_add_value_alias_() (*enum.Enum* 的方法), 326
align (*ctypes.Structure* 的屬性), 850
anonymous (*ctypes.Structure* 的屬性), 851
_asdict() (*collections.somenamedtuple* 的方法), 275
_b_base_ (*ctypes._CData* 的屬性), 846
_b_needsfree_ (*ctypes._CData* 的屬性), 846
_callmethod() (*multiprocessing.managers.BaseProxy* 的方法), 1009
_CData (*ctypes* 中的類F), 845
_CFuncPtr (*ctypes* 中的類F), 838
_clear_internal_caches() (於 *sys* 模組中), 1921
_clear_type_cache() (於 *sys* 模組中), 1921
_current_exceptions() (於 *sys* 模組中), 1921
_current_frames() (於 *sys* 模組中), 1921
_debugmallocstats() (於 *sys* 模組中), 1922
_emscripten_info (於 *sys* 模組中), 1923
_enablelegacywindowsfsencoding() (於 *sys* 模組中), 1943
_enter_task() (於 *asyncio* 模組中), 1163
_exit() (於 *os* 模組中), 717
_Feature (*__future__* 中的類F), 2011
_field_defaults (*collections.somenamedtuple* 的屬性), 275
_field_types (*ast.AST* 的屬性), 2096
_fields (*ast.AST* 的屬性), 2096
_fields (*collections.somenamedtuple* 的屬性), 275
fields (*ctypes.Structure* 的屬性), 850
_flush() (*wsgiref.handlers.BaseHandler* 的方法), 1431
_for_archive() (*zipfile.ZipInfo* 的方法), 591
_generate_next_value_() (*enum.Enum* 的方法), 323
_get_child_mock() (*unittest.mock.Mock* 的方法), 1783

_get_preferred_schemes() (於 *sysconfig* 模組中), 1955
_getframe() (於 *sys* 模組中), 1930
_getframemodulename() (於 *sys* 模組中), 1931
_getvalue() (*multiprocessing.managers.BaseProxy* 的方法), 1010
_handle (*ctypes.PyDLL* 的屬性), 837
ignore (*enum.Enum* 的屬性), 323
_incompatible_extension_module_restrictions (於 *importlib.util* 模組中), 2075
_is_gil_enabled() (於 *sys* 模組中), 1934
_is_immortal() (於 *sys* 模組中), 1936
_is_interned() (於 *sys* 模組中), 1936
_jit (於 *sys* 模組中), 1935
_jit.is_active() (於 *sys* 模組中), 1935
_jit.is_available() (於 *sys* 模組中), 1935
_jit.is_enabled() (於 *sys* 模組中), 1935
layout (*ctypes.Structure* 的屬性), 850
_leave_task() (於 *asyncio* 模組中), 1163
length (*ctypes.Array* 的屬性), 853
_locale
    module (模組), 1569
_log (*logging.LoggerAdapter* 的屬性), 773
_make() (*collections.somenamedtuple* 的類🄵方法), 274
_makeResult() (*unittest.TextTestRunner* 的方法), 1773
missing() (*enum.Enum* 的方法), 324
_name (*ctypes.PyDLL* 的屬性), 837
name (*enum.Enum* 的屬性), 323
_numeric_repr_() (*enum.Flag* 的方法), 328
_objects (*ctypes._CData* 的屬性), 846
order (*enum.Enum* 的屬性), 323
pack (*ctypes.Structure* 的屬性), 850
_parse() (*gettext.NullTranslations* 的方法), 1563
_Pointer (*ctypes* 中的類🄵), 853
_register_task() (於 *asyncio* 模組中), 1163
_replace() (*collections.somenamedtuple* 的方法), 275
_setroot() (*xml.etree.ElementTree.ElementTree* 的方法), 1379
_SimpleCData (*ctypes* 中的類🄵), 846
_structure() (於 *email.iterators* 模組中), 1319
_thread
    module, 1071
_tkinter
    module, 1582
type (*ctypes._Pointer* 的屬性), 853
type (*ctypes.Array* 的屬性), 853
_unregister_task() (於 *asyncio* 模組中), 1163
value (*enum.Enum* 的屬性), 323
_write() (*wsgiref.handlers.BaseHandler* 的方法), 1431
_xoptions (於 *sys* 模組中), 1946
{} (花括號)
    於字串格式化, 125
    於格式化字串常數中, 63
    於正規表示式中, 140
| (垂直🄵)
    operator (運算子), 40
    於正規表示式中, 142
~ (波浪號)
    home directory expansion (家目🄵展開), 470
    operator (運算子), 40
事件 (元件), 1589
二進制模式, 24
環境變數
    BROWSER, 1423, 1424
    COLUMNS, 935
    COMSPEC, 725, 1049
    DISPLAY, 1581
    HOME, 470, 1581
    HOMEDRIVE, 470
    HOMEPATH, 470
    IDLESTARTUP, 1624
    LANG, 1561, 1562, 1569, 1573
    LANGUAGE, 1561, 1562
    LC_ALL, 1561, 1562
    LC_MESSAGES, 1561, 1562
    LINES, 930, 935
    LOGNAME, 672, 926
    MANPAGER, 1720
    no_proxy, 1439
    PAGER, 1720
    PATH, 467, 497, 716, 717, 722, 723, 732, 1048, 1423, 1907, 2044
    PATHEXT, 497
    PYTHON_CONTEXT_AWARE_WARNINGS, 1926, 1969
    PYTHON_CPU_COUNT, 731, 996
    PYTHON_DOM, 1385
    PYTHON_GIL, 1926, 2234
    PYTHON_JIT, 1935
    PYTHON_THREAD_INHERIT_CONTEXT, 1926
    PYTHONASYNCIODEBUG, 1137, 1171, 1722
    PYTHONBREAKPOINT, 9, 1922
    PYTHONCASEOK, 33
    PYTHONCOERCECLOCALE, 669
    PYTHONDEVMODE, 1721
    PYTHONDONTWRITEBYTECODE, 1923
    PYTHONFAULTHANDLER, 1722, 1865
    PYTHONHOME, 1846, 2091, 2092
    PYTHONINTMAXSTRDIGITS, 106, 1934
    PYTHONIOENCODING, 668, 1943
    PYTHONLEGACYWINDOWSFSENCODING, 1943
    PYTHONLEGACYWINDOWSSTDIO, 1943
    PYTHONMALLOC, 1722
    PYTHONNOUSERSITE, 2045
    PYTHONPATH, 1846, 1937, 2091
    PYTHONPLATLIBDIR, 2091
    PYTHONPYCACHEPREFIX, 1923
    PYTHONSAFEPATH, 1937, 2225
    PYTHONSTARTUP, 183, 1076, 1624, 1934, 2045
    PYTHONTRACEMALLOC, 1893, 1898
    PYTHONTZPATH, 252, 255
    PYTHONUNBUFFERED, 1943

PYTHONUSERBASE, 2045
PYTHONUSERSITE, 1846
PYTHONUTF8, 669, 1943
PYTHONWARNDEFAULTENCODING, 735
PYTHONWARNINGS, 1721, 1964
SOURCE_DATE_EPOCH, 2146, 2148
SSLKEYLOGFILE, 1204, 1205
SystemRoot, 1051
TEMP, 486
TERM, 934
TMP, 486
TMPDIR, 486
TZ, 756, 757
USER, 926
USERNAME, 470, 672, 926
USERPROFILE, 470
空格
於字串格式化, 127
紀元秒數, 748
視窗管理器（元件）, 1587

A

-a
ast 命令列選項, 2129
pickletools 命令列選項, 2174
A (於 *re* 模組中), 146
a2b_base64() (於 *binascii* 模組中), 1356
a2b_hex() (於 *binascii* 模組中), 1357
a2b_qp() (於 *binascii* 模組中), 1357
a2b_uu() (於 *binascii* 模組中), 1356
a85decode() (於 *base64* 模組中), 1354
a85encode() (於 *base64* 模組中), 1354
A_ALTCHARSET (於 *curses* 模組中), 943
A_ATTRIBUTES (於 *curses* 模組中), 944
A_BLINK (於 *curses* 模組中), 943
A_BOLD (於 *curses* 模組中), 943
A_CHARTEXT (於 *curses* 模組中), 944
A_COLOR (於 *curses* 模組中), 944
A_DIM (於 *curses* 模組中), 943
A_HORIZONTAL (於 *curses* 模組中), 943
A_INVIS (於 *curses* 模組中), 943
A_ITALIC (於 *curses* 模組中), 943
A_LEFT (於 *curses* 模組中), 943
A_LOW (於 *curses* 模組中), 943
A_NORMAL (於 *curses* 模組中), 943
A_PROTECT (於 *curses* 模組中), 943
A_REVERSE (於 *curses* 模組中), 943
A_RIGHT (於 *curses* 模組中), 943
A_STANDOUT (於 *curses* 模組中), 943
A_TOP (於 *curses* 模組中), 943
A_UNDERLINE (於 *curses* 模組中), 943
A_VERTICAL (於 *curses* 模組中), 943
abc
module, 1994
ABC (*abc* 中的類F), 1994
ABCMeta (*abc* 中的類F), 1994
ABDAY_1 (於 *locale* 模組中), 1571
ABDAY_2 (於 *locale* 模組中), 1571
ABDAY_3 (於 *locale* 模組中), 1571
ABDAY_4 (於 *locale* 模組中), 1571
ABDAY_5 (於 *locale* 模組中), 1571
ABDAY_6 (於 *locale* 模組中), 1571
ABDAY_7 (於 *locale* 模組中), 1571
abiflags (於 *sys* 模組中), 1919
ABMON_1 (於 *locale* 模組中), 1572
ABMON_2 (於 *locale* 模組中), 1572
ABMON_3 (於 *locale* 模組中), 1572
ABMON_4 (於 *locale* 模組中), 1572
ABMON_5 (於 *locale* 模組中), 1572
ABMON_6 (於 *locale* 模組中), 1572
ABMON_7 (於 *locale* 模組中), 1572
ABMON_8 (於 *locale* 模組中), 1572
ABMON_9 (於 *locale* 模組中), 1572
ABMON_10 (於 *locale* 模組中), 1572
ABMON_11 (於 *locale* 模組中), 1572
ABMON_12 (於 *locale* 模組中), 1572
ABORT (於 *tkinter.messagebox* 模組中), 1597
abort() (*asyncio.Barrier* 的方法), 1110
abort() (*asyncio.DatagramTransport* 的方法), 1150
abort() (*asyncio.WriteTransport* 的方法), 1149
abort() (*ftplib.FTP* 的方法), 1476
abort() (*threading.Barrier* 的方法), 981
abort() (於 *os* 模組中), 716
abort_clients() (*asyncio.Server* 的方法), 1139
ABORTRETRYIGNORE (於 *tkinter.messagebox* 模組中), 1597
above() (*curses.panel.Panel* 的方法), 958
ABOVE_NORMAL_PRIORITY_CLASS (於 *subprocess* 模組中), 1056
abs()
built-in function, 8
abs() (*decimal.Context* 的方法), 367
abs() (於 *operator* 模組中), 436
absolute() (*pathlib.Path* 的方法), 456
absolute_import (於 *__future__* 模組中), 2011
AbsoluteLinkError, 605
AbsolutePathError, 604
abspath() (於 *os.path* 模組中), 469
abstract base class（抽象基底類F）, **2227**
AbstractAsyncContextManager (*contextlib* 中的類F), 1980
AbstractBasicAuthHandler (*urllib.request* 中的類F), 1439
abstractclassmethod() (於 *abc* 模組中), 1997
AbstractContextManager (*contextlib* 中的類F), 1980
AbstractDigestAuthHandler (*urllib.request* 中的類F), 1440
AbstractEventLoop (*asyncio* 中的類F), 1141
AbstractEventLoopPolicy (*asyncio* 中的類F), 1160
abstractmethod() (於 *abc* 模組中), 1996
abstractproperty() (於 *abc* 模組中), 1997
AbstractSet (*typing* 中的類F), 1716
abstractstaticmethod() (於 *abc* 模組中), 1997

accept() (*multiprocessing.connection.Listener* 的方法), 1014
accept() (*socket.socket* 的方法), 1191
access() (於 *os* 模組中), 690
accumulate() (於 *itertools* 模組中), 411
ACK (於 *curses.ascii* 模組中), 955
aclose() (*contextlib.AsyncExitStack* 的方法), 1988
aclosing() (於 *contextlib* 模組中), 1982
acos() (於 *cmath* 模組中), 351
acos() (於 *math* 模組中), 347
acosh() (於 *cmath* 模組中), 351
acosh() (於 *math* 模組中), 347
acquire() (*_thread.lock* 的方法), 1073
acquire() (*asyncio.Condition* 的方法), 1108
acquire() (*asyncio.Lock* 的方法), 1106
acquire() (*asyncio.Semaphore* 的方法), 1109
acquire() (*logging.Handler* 的方法), 766
acquire() (*multiprocessing.Lock* 的方法), 1000
acquire() (*multiprocessing.RLock* 的方法), 1000
acquire() (*threading.Condition* 的方法), 977
acquire() (*threading.Lock* 的方法), 974
acquire() (*threading.RLock* 的方法), 975
acquire() (*threading.Semaphore* 的方法), 978
ACS_BBSS (於 *curses* 模組中), 950
ACS_BLOCK (於 *curses* 模組中), 950
ACS_BOARD (於 *curses* 模組中), 950
ACS_BSBS (於 *curses* 模組中), 950
ACS_BSSB (於 *curses* 模組中), 950
ACS_BSSS (於 *curses* 模組中), 950
ACS_BTEE (於 *curses* 模組中), 950
ACS_BULLET (於 *curses* 模組中), 950
ACS_CKBOARD (於 *curses* 模組中), 950
ACS_DARROW (於 *curses* 模組中), 950
ACS_DEGREE (於 *curses* 模組中), 950
ACS_DIAMOND (於 *curses* 模組中), 950
ACS_GEQUAL (於 *curses* 模組中), 950
ACS_HLINE (於 *curses* 模組中), 950
ACS_LANTERN (於 *curses* 模組中), 950
ACS_LARROW (於 *curses* 模組中), 950
ACS_LEQUAL (於 *curses* 模組中), 950
ACS_LLCORNER (於 *curses* 模組中), 950
ACS_LRCORNER (於 *curses* 模組中), 951
ACS_LTEE (於 *curses* 模組中), 951
ACS_NEQUAL (於 *curses* 模組中), 951
ACS_PI (於 *curses* 模組中), 951
ACS_PLMINUS (於 *curses* 模組中), 951
ACS_PLUS (於 *curses* 模組中), 951
ACS_RARROW (於 *curses* 模組中), 951
ACS_RTEE (於 *curses* 模組中), 951
ACS_S1 (於 *curses* 模組中), 951
ACS_S3 (於 *curses* 模組中), 951
ACS_S7 (於 *curses* 模組中), 951
ACS_S9 (於 *curses* 模組中), 951
ACS_SBBS (於 *curses* 模組中), 951
ACS_SBSB (於 *curses* 模組中), 951
ACS_SBSS (於 *curses* 模組中), 951
ACS_SSBB (於 *curses* 模組中), 951
ACS_SSBS (於 *curses* 模組中), 951
ACS_SSSB (於 *curses* 模組中), 951
ACS_SSSS (於 *curses* 模組中), 952
ACS_STERLING (於 *curses* 模組中), 952
ACS_TTEE (於 *curses* 模組中), 952
ACS_UARROW (於 *curses* 模組中), 952
ACS_ULCORNER (於 *curses* 模組中), 952
ACS_URCORNER (於 *curses* 模組中), 952
ACS_VLINE (於 *curses* 模組中), 952
Action (*argparse* 中的類🄵), 873
action (*optparse.Option* 的屬性), 912
ACTIONS (*optparse.Option* 的屬性), 923
activate_stack_trampoline() (於 *sys* 模組中), 1942
active_children() (於 *multiprocessing* 模組中), 995
active_count() (於 *threading* 模組中), 966
actual() (*tkinter.font.Font* 的方法), 1592
Add (*ast* 中的類🄵), 2102
add() (*decimal.Context* 的方法), 368
add() (*frozenset* 的方法), 91
add() (*graphlib.TopologicalSorter* 的方法), 335
add() (*mailbox.Mailbox* 的方法), 1330
add() (*mailbox.Maildir* 的方法), 1334
add() (*pstats.Stats* 的方法), 1882
add() (*tarfile.TarFile* 的方法), 609
add() (*tkinter.ttk.Notebook* 的方法), 1605
add() (於 *operator* 模組中), 436
add_alias() (於 *email.charset* 模組中), 1315
add_alternative() (*email.message.EmailMessage* 的方法), 1271
add_argument() (*argparse.ArgumentParser* 的方法), 864
add_argument_group() (*argparse.ArgumentParser* 的方法), 880
add_attachment() (*email.message.EmailMessage* 的方法), 1271
add_cgi_vars() (*wsgiref.handlers.BaseHandler* 的方法), 1431
add_charset() (於 *email.charset* 模組中), 1315
add_codec() (於 *email.charset* 模組中), 1315
add_cookie_header() (*http.cookiejar.CookieJar* 的方法), 1523
add_dll_directory() (於 *os* 模組中), 716
add_done_callback() (*asyncio.Future* 的方法), 1145
add_done_callback() (*asyncio.Task* 的方法), 1095
add_done_callback() (*concurrent.futures.Future* 的方法), 1037
add_fallback() (*gettext.NullTranslations* 的方法), 1563
add_flag() (*mailbox.Maildir* 的方法), 1334
add_flag() (*mailbox.MaildirMessage* 的方法), 1340
add_flag() (*mailbox.mboxMessage* 的方法), 1342
add_flag() (*mailbox.MMDFMessage* 的方法), 1346
add_folder() (*mailbox.Maildir* 的方法), 1333
add_folder() (*mailbox.MH* 的方法), 1336
add_get_handler() (*email.contentmanager.ContentManager*

的方法), 1292
add_handler() (*urllib.request.OpenerDirector* 的方法), 1442
add_header() (*email.message.EmailMessage* 的方法), 1267
add_header() (*email.message.Message* 的方法), 1304
add_header() (*urllib.request.Request* 的方法), 1441
add_header() (*wsgiref.headers.Headers* 的方法), 1428
add_history() (於 *readline* 模組中), 182
add_label() (*mailbox.BabylMessage* 的方法), 1344
add_mutually_exclusive_group() (*argparse.ArgumentParser* 的方法), 881
add_note() (*BaseException* 的方法), 110
add_option() (*optparse.OptionParser* 的方法), 910
add_parent() (*urllib.request.BaseHandler* 的方法), 1443
add_password() (*urllib.request.HTTPPasswordMgr* 的方法), 1445
add_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 的方法), 1446
add_reader() (*asyncio.loop* 的方法), 1131
add_related() (*email.message.EmailMessage* 的方法), 1271
add_section() (*configparser.ConfigParser* 的方法), 640
add_section() (*configparser.RawConfigParser* 的方法), 643
add_sequence() (*mailbox.MHMessage* 的方法), 1343
add_set_handler() (*email.contentmanager.ContentManager* 的方法), 1292
add_signal_handler() (*asyncio.loop* 的方法), 1134
add_subparsers() (*argparse.ArgumentParser* 的方法), 877
add_type() (*mimetypes.MimeTypes* 的方法), 1351
add_type() (於 *mimetypes* 模組中), 1349
add_unredirected_header() (*urllib.request.Request* 的方法), 1441
add_writer() (*asyncio.loop* 的方法), 1131
addAsyncCleanup() (*unittest.IsolatedAsyncioTestCase* 的方法), 1765
addaudithook() (於 *sys* 模組中), 1919
addch() (*curses.window* 的方法), 936
addClassCleanup() (*unittest.TestCase* 的類🄵方法), 1765
addCleanup() (*unittest.TestCase* 的方法), 1764
addcomponent() (*turtle.Shape* 的方法), 1660
addDuration() (*unittest.TestResult* 的方法), 1772
addError() (*unittest.TestResult* 的方法), 1771
addExpectedFailure() (*unittest.TestResult* 的方法), 1772
addFailure() (*unittest.TestResult* 的方法), 1771
addfile() (*tarfile.TarFile* 的方法), 609
addFilter() (*logging.Handler* 的方法), 766
addFilter() (*logging.Logger* 的方法), 764
addHandler() (*logging.Logger* 的方法), 765
addinfourl (*urllib.response* 中的類🄵), 1452
addLevelName() (於 *logging* 模組中), 775
addModuleCleanup() (於 *unittest* 模組中), 1775
addnstr() (*curses.window* 的方法), 936
AddPackagePath() (於 *modulefinder* 模組中), 2055
addr_spec (*email.headerregistry.Address* 的屬性), 1291
Address (*email.headerregistry* 中的類🄵), 1290
address (*email.headerregistry.SingleAddressHeader* 的屬性), 1288
address (*multiprocessing.connection.Listener* 的屬性), 1014
address (*multiprocessing.managers.BaseManager* 的屬性), 1005
address_exclude() (*ipaddress.IPv4Network* 的方法), 1551
address_exclude() (*ipaddress.IPv6Network* 的方法), 1553
address_family (*socketserver.BaseServer* 的屬性), 1505
address_string() (*http.server.BaseHTTPRequestHandler* 的方法), 1515
addresses (*email.headerregistry.AddressHeader* 的屬性), 1288
addresses (*email.headerregistry.Group* 的屬性), 1291
AddressHeader (*email.headerregistry* 中的類🄵), 1288
addressof() (於 *ctypes* 模組中), 842
AddressValueError, 1556
addshape() (於 *turtle* 模組中), 1657
addsitedir() (於 *site* 模組中), 2045
addSkip() (*unittest.TestResult* 的方法), 1772
addstr() (*curses.window* 的方法), 936
addSubTest() (*unittest.TestResult* 的方法), 1772
addSuccess() (*unittest.TestResult* 的方法), 1771
addTest() (*unittest.TestSuite* 的方法), 1767
addTests() (*unittest.TestSuite* 的方法), 1767
addTypeEqualityFunc() (*unittest.TestCase* 的方法), 1762
addUnexpectedSuccess() (*unittest.TestResult* 的方法), 1772
adjust_int_max_str_digits() (於 *test.support* 模組中), 1845
adjusted() (*decimal.Decimal* 的方法), 358
adler32() (於 *zlib* 模組中), 574
AF_ALG (於 *socket* 模組中), 1181
AF_BLUETOOTH (於 *socket* 模組中), 1181
AF_CAN (於 *socket* 模組中), 1179
AF_DIVERT (於 *socket* 模組中), 1180
AF_HYPERV (於 *socket* 模組中), 1183
AF_INET (於 *socket* 模組中), 1178
AF_INET6 (於 *socket* 模組中), 1178
AF_LINK (於 *socket* 模組中), 1181
AF_PACKET (於 *socket* 模組中), 1180
AF_QIPCRTR (於 *socket* 模組中), 1183
AF_RDS (於 *socket* 模組中), 1181
AF_UNIX (於 *socket* 模組中), 1178
AF_UNSPEC (於 *socket* 模組中), 1178

AF_VSOCK (於 *socket* 模組中), 1181
aifc
    module, 2219
aiter()
    built-in function, 8
alarm() (於 *signal* 模組中), 1251
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (於 *ssl* 模組中), 1213
ALERT_DESCRIPTION_INTERNAL_ERROR (於 *ssl* 模組中), 1213
AlertDescription (*ssl* 中的類🄵), 1213
algorithm (*sys.hash_info* 的屬性), 1933
algorithms_available (於 *hashlib* 模組中), 653
algorithms_guaranteed (於 *hashlib* 模組中), 653
alias (*ast* 中的類🄵), 2109
alias (*pdb command*), 1876
Alias (🄵名)
    Generic (泛型), 97
alignment() (於 *ctypes* 模組中), 842
alive (*weakref.finalize* 的屬性), 299
all()
    built-in function, 8
ALL_COMPLETED (於 *asyncio* 模組中), 1090
ALL_COMPLETED (於 *concurrent.futures* 模組中), 1038
all_errors (於 *ftplib* 模組中), 1479
all_features (於 *xml.sax.handler* 模組中), 1403
all_frames (*tracemalloc.Filter* 的屬性), 1899
all_properties (於 *xml.sax.handler* 模組中), 1403
all_suffixes() (於 *importlib.machinery* 模組中), 2068
all_tasks() (於 *asyncio* 模組中), 1094
allocate_lock() (於 *_thread* 模組中), 1072
ALLOW_MISSING (於 *os.path* 模組中), 473
allow_reuse_address (*socketserver.BaseServer* 的屬性), 1506
allowed_domains() (*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
alt() (於 *curses.ascii* 模組中), 958
ALT_DIGITS (於 *locale* 模組中), 1573
--altinstall
    ensurepip 命令列選項, 1904
altsep (於 *os* 模組中), 732
altzone (於 *time* 模組中), 759
ALWAYS_EQ (於 *test.support* 模組中), 1838
ALWAYS_TYPED_ACTIONS (*optparse.Option* 的屬性), 924
AmbiguousOptionError, 925
AMPER (於 *token* 模組中), 2136
AMPEREQUAL (於 *token* 模組中), 2138
Anchor (*importlib.resources* 中的類🄵), 2079
anchor (*pathlib.PurePath* 的屬性), 448
and
    operator (運算子), 37, 38
And (*ast* 中的類🄵), 2102
and_() (於 *operator* 模組中), 436
android_ver() (於 *platform* 模組中), 808
anext()
    built-in function, 8
AnnAssign (*ast* 中的類🄵), 2107
--annotate
    pickletools 命令列選項, 2174
annotate function (🄵釋函式), **2227**
Annotated (於 *typing* 模組中), 1685
annotation (*inspect.Parameter* 的屬性), 2024
ANNOTATION (*symtable.SymbolTableType* 的屬性), 2130
annotationlib
    module, 2035
annotations (於 *__future__* 模組中), 2011
annotations_to_string() (於 *annotationlib* 模組中), 2038
annotation (🄵記)
    type annotation(型🄵🄵記); type hint(型🄵提示), 97
annotation (🄵釋), **2227**
answer_challenge() (於 *multiprocessing.connection* 模組中), 1013
anticipate_failure() (於 *test.support* 模組中), 1842
Any (於 *typing* 模組中), 1678
ANY (於 *unittest.mock* 模組中), 1809
any()
    built-in function, 9
ANY_CONTIGUOUS (*inspect.BufferFlags* 的屬性), 2034
AnyStr (於 *typing* 模組中), 1678
api_version (於 *sys* 模組中), 1945
apilevel (於 *sqlite3* 模組中), 536
apop() (*poplib.POP3* 的方法), 1481
append() (*array.array* 的方法), 294
append() (*collections.deque* 的方法), 269
append() (*email.header.Header* 的方法), 1312
append() (*imaplib.IMAP4* 的方法), 1484
append() (*sequence* 的方法), 48
append() (*xml.etree.ElementTree.Element* 的方法), 1377
append_history_file() (於 *readline* 模組中), 181
appendChild() (*xml.dom.Node* 的方法), 1388
appendleft() (*collections.deque* 的方法), 269
AppleFrameworkLoader (*importlib.machinery* 中的類🄵), 2073
application_uri() (於 *wsgiref.util* 模組中), 1426
apply() (*multiprocessing.pool.Pool* 的方法), 1011
apply_async() (*multiprocessing.pool.Pool* 的方法), 1011
apply_defaults() (*inspect.BoundArguments* 的方法), 2026
APRIL (於 *calendar* 模組中), 261
architecture() (於 *platform* 模組中), 804
archive (*zipimport.zipimporter* 的屬性), 2052
AREGTYPE (於 *tarfile* 模組中), 605
aRepr (於 *reprlib* 模組中), 316
arg (*ast* 中的類🄵), 2121
argparse
    module, 855
args (*BaseException* 的屬性), 110

args (*functools.partial* 的屬性), 435
args (*inspect.BoundArguments* 的屬性), 2026
args (*pdb command*), 1874
args (*subprocess.CompletedProcess* 的屬性), 1045
args (*subprocess.Popen* 的屬性), 1054
args (*typing.ParamSpec* 的屬性), 1696
args_from_interpreter_flags() (於 *test.support* 模組中), 1840
argtypes (*ctypes._CFuncPtr* 的屬性), 839
ArgumentDefaultsHelpFormatter (*argparse* 中的類🄵), 859
ArgumentError, 853, 885
ArgumentParser (*argparse* 中的類🄵), 856
arguments (*ast* 中的類🄵), 2121
arguments (*inspect.BoundArguments* 的屬性), 2026
ArgumentTypeError, 885
argument（引數）, **2227**
argv (於 *sys* 模組中), 1920
ArithmeticError, 110
arithmetic（算術）, 38
array
    module, 293
array (*array* 中的類🄵), 294
Array (*ctypes* 中的類🄵), 852
Array() (*multiprocessing.managers.SyncManager* 的方法), 1006
ARRAY() (於 *ctypes* 模組中), 853
Array() (於 *multiprocessing* 模組中), 1002
Array() (於 *multiprocessing.sharedctypes* 模組中), 1003
arraysize (*sqlite3.Cursor* 的屬性), 550
arrays（陣列）, 293
array（陣列）
    模組, 66
as_bytes() (*email.message.EmailMessage* 的方法), 1265
as_bytes() (*email.message.Message* 的方法), 1301
as_completed() (於 *asyncio* 模組中), 1091
as_completed() (於 *concurrent.futures* 模組中), 1038
as_digested_dict (*compression.zstd.ZstdDict* 的屬性), 569
as_file() (於 *importlib.resources* 模組中), 2079
as_integer_ratio(), 381
as_integer_ratio() (*decimal.Decimal* 的方法), 358
as_integer_ratio() (*float* 的方法), 42
as_integer_ratio() (*fractions.Fraction* 的方法), 382
as_integer_ratio() (*int* 的方法), 42
as_posix() (*pathlib.PurePath* 的方法), 450
as_string() (*email.message.EmailMessage* 的方法), 1265
as_string() (*email.message.Message* 的方法), 1301
as_tuple() (*decimal.Decimal* 的方法), 358
as_undigested_dict (*compression.zstd.ZstdDict* 的屬性), 569
as_uri() (*pathlib.Path* 的方法), 455
ASCII (於 *re* 模組中), 146
ascii()
    built-in function, 9
ascii() (於 *curses.ascii* 模組中), 957
ascii_letters (於 *string* 模組中), 123
ascii_lowercase (於 *string* 模組中), 123
ascii_uppercase (於 *string* 模組中), 123
asctime() (於 *time* 模組中), 748
asdict() (於 *dataclasses* 模組中), 1974
asin() (於 *cmath* 模組中), 351
asin() (於 *math* 模組中), 347
asinh() (於 *cmath* 模組中), 351
asinh() (於 *math* 模組中), 347
askcolor() (於 *tkinter.colorchooser* 模組中), 1591
askdirectory() (於 *tkinter.filedialog* 模組中), 1594
askfloat() (於 *tkinter.simpledialog* 模組中), 1592
askinteger() (於 *tkinter.simpledialog* 模組中), 1592
askokcancel() (於 *tkinter.messagebox* 模組中), 1596
askopenfile() (於 *tkinter.filedialog* 模組中), 1593
askopenfilename() (於 *tkinter.filedialog* 模組中), 1594
askopenfilenames() (於 *tkinter.filedialog* 模組中), 1594
askopenfiles() (於 *tkinter.filedialog* 模組中), 1593
askquestion() (於 *tkinter.messagebox* 模組中), 1596
askretrycancel() (於 *tkinter.messagebox* 模組中), 1596
asksaveasfile() (於 *tkinter.filedialog* 模組中), 1593
asksaveasfilename() (於 *tkinter.filedialog* 模組中), 1594
askstring() (於 *tkinter.simpledialog* 模組中), 1592
askyesno() (於 *tkinter.messagebox* 模組中), 1596
askyesnocancel() (於 *tkinter.messagebox* 模組中), 1597
assert
    statement（陳述式）, 111
Assert (*ast* 中的類🄵), 2108
assert_any_await() (*unittest.mock.AsyncMock* 的方法), 1790
assert_any_call() (*unittest.mock.Mock* 的方法), 1781
assert_awaited() (*unittest.mock.AsyncMock* 的方法), 1789
assert_awaited_once() (*unittest.mock.AsyncMock* 的方法), 1789
assert_awaited_once_with() (*unittest.mock.AsyncMock* 的方法), 1790
assert_awaited_with() (*unittest.mock.AsyncMock* 的方法), 1789
assert_called() (*unittest.mock.Mock* 的方法), 1780
assert_called_once() (*unittest.mock.Mock* 的方法), 1780
assert_called_once_with() (*unittest.mock.Mock* 的方法), 1780
assert_called_with() (*unittest.mock.Mock* 的方法), 1780
assert_has_awaits() (*unittest.mock.AsyncMock* 的方法), 1790

assert_has_calls() (*unittest.mock.Mock* 的方法), 1781
assert_never() (於 *typing* 模組中), 1706
assert_not_awaited() (*unittest.mock.AsyncMock* 的方法), 1791
assert_not_called() (*unittest.mock.Mock* 的方法), 1781
assert_python_failure() (於 *test.support.script_helper* 模組中), 1847
assert_python_ok() (於 *test.support.script_helper* 模組中), 1847
assert_type() (於 *typing* 模組中), 1706
assertAlmostEqual() (*unittest.TestCase* 的方法), 1761
assertCountEqual() (*unittest.TestCase* 的方法), 1762
assertDictEqual() (*unittest.TestCase* 的方法), 1763
assertEndsWith() (*unittest.TestCase* 的方法), 1762
assertEqual() (*unittest.TestCase* 的方法), 1757
assertFalse() (*unittest.TestCase* 的方法), 1757
assertGreater() (*unittest.TestCase* 的方法), 1761
assertGreaterEqual() (*unittest.TestCase* 的方法), 1761
assertHasAttr() (*unittest.TestCase* 的方法), 1762
assertIn() (*unittest.TestCase* 的方法), 1758
assertInBytecode() (*test.support.bytecode_helper.BytecodeTestCase* 的方法), 1848
AssertionError, 111
assertIs() (*unittest.TestCase* 的方法), 1757
assertIsInstance() (*unittest.TestCase* 的方法), 1758
assertIsNone() (*unittest.TestCase* 的方法), 1758
assertIsNot() (*unittest.TestCase* 的方法), 1757
assertIsNotNone() (*unittest.TestCase* 的方法), 1758
assertIsSubclass() (*unittest.TestCase* 的方法), 1758
assertLess() (*unittest.TestCase* 的方法), 1761
assertLessEqual() (*unittest.TestCase* 的方法), 1761
assertListEqual() (*unittest.TestCase* 的方法), 1763
assertLogs() (*unittest.TestCase* 的方法), 1760
assertMultiLineEqual() (*unittest.TestCase* 的方法), 1763
assertNoLogs() (*unittest.TestCase* 的方法), 1760
assertNotAlmostEqual() (*unittest.TestCase* 的方法), 1761
assertNotEndsWith() (*unittest.TestCase* 的方法), 1762
assertNotEqual() (*unittest.TestCase* 的方法), 1757
assertNotHasAttr() (*unittest.TestCase* 的方法), 1762
assertNotIn() (*unittest.TestCase* 的方法), 1758
assertNotInBytecode() (*test.support.bytecode_helper.BytecodeTestCase* 的方法), 1848
assertNotIsInstance() (*unittest.TestCase* 的方法), 1758
assertNotIsSubclass() (*unittest.TestCase* 的方法), 1758
assertNotRegex() (*unittest.TestCase* 的方法), 1762
assertNotStartsWith() (*unittest.TestCase* 的方法), 1762
assertRaises() (*unittest.TestCase* 的方法), 1758
assertRaisesRegex() (*unittest.TestCase* 的方法), 1759
assertRegex() (*unittest.TestCase* 的方法), 1762
assertSequenceEqual() (*unittest.TestCase* 的方法), 1763
assertSetEqual() (*unittest.TestCase* 的方法), 1763
assertStartsWith() (*unittest.TestCase* 的方法), 1762
assertTrue() (*unittest.TestCase* 的方法), 1757
assertTupleEqual() (*unittest.TestCase* 的方法), 1763
assertWarns() (*unittest.TestCase* 的方法), 1759
assertWarnsRegex() (*unittest.TestCase* 的方法), 1760
Assign (*ast* 中的類🄵), 2107
assignment (賦值)
    slice (切片), 48
    subscript (下標), 48
assume_default_colors() (於 *curses* 模組中), 929
ast
    module, 2093
AST (*ast* 中的類🄵), 2096
ast 命令列選項
    -a, 2129
    --feature-version, 2129
    -h, 2129
    --help, 2129
    -i, 2129
    --include-attributes, 2129
    --indent, 2129
    -m, 2129
    --mode, 2129
    --no-type-comments, 2129
    -O, 2129
    --optimize, 2129
    --show-empty, 2129
astimezone() (*datetime.datetime* 的方法), 230
astuple() (於 *dataclasses* 模組中), 1974
AsyncContextDecorator (*contextlib* 中的類🄵), 1986
AsyncContextManager (*typing* 中的類🄵), 1719
asynccontextmanager() (於 *contextlib* 模組中), 1981
AsyncExitStack (*contextlib* 中的類🄵), 1988
AsyncFor (*ast* 中的類🄵), 2124
AsyncFunctionDef (*ast* 中的類🄵), 2124
AsyncGenerator (*collections.abc* 中的類🄵), 285
AsyncGenerator (*typing* 中的類🄵), 1718
AsyncGeneratorType (於 *types* 模組中), 305
asynchat
    module, 2219
asynchronous context manager (非同步情境管理器), **2228**

asynchronous generator iterator（非同步F生器F代器）, **2228**
asynchronous generator（非同步F生器）, **2228**
asynchronous iterable（非同步可F代物件）, **2228**
asynchronous iterator（非同步F代器）, **2228**
asyncio
    module, 1075
asyncio.subprocess.DEVNULL（F建變數）, 1112
asyncio.subprocess.PIPE（F建變數）, 1112
asyncio.subprocess.Process（F建類F）, 1113
asyncio.subprocess.STDOUT（F建變數）, 1112
AsyncIterable（*collections.abc* 中的類F）, 284
AsyncIterable（*typing* 中的類F）, 1718
AsyncIterator（*collections.abc* 中的類F）, 285
AsyncIterator（*typing* 中的類F）, 1718
AsyncMock（*unittest.mock* 中的類F）, 1788
asyncore
    module, 2219
AsyncResult（*multiprocessing.pool* 中的類F）, 1012
asyncSetUp()（*unittest.IsolatedAsyncioTestCase* 的方法）, 1765
asyncTearDown()（*unittest.IsolatedAsyncioTestCase* 的方法）, 1765
AsyncWith（*ast* 中的類F）, 2124
AT（於 *token* 模組中）, 2138
at_eof()（*asyncio.StreamReader* 的方法）, 1101
atan()（於 *cmath* 模組中）, 351
atan()（於 *math* 模組中）, 347
atan2()（於 *math* 模組中）, 347
atanh()（於 *cmath* 模組中）, 351
atanh()（於 *math* 模組中）, 347
ATEQUAL（於 *token* 模組中）, 2138
atexit
    module, 1999
atexit（*weakref.finalize* 的屬性）, 299
atof()（於 *locale* 模組中）, 1575
atoi()（於 *locale* 模組中）, 1575
attach()（*email.message.Message* 的方法）, 1302
attach_mock()（*unittest.mock.Mock* 的方法）, 1782
attached thread state, **2228**
attempted（*doctest.TestResults* 的屬性）, 1741
AttlistDeclHandler()（*xml.parsers.expat.xmlparser* 的方法）, 1415
attrgetter()（於 *operator* 模組中）, 438
attrib（*xml.etree.ElementTree.Element* 的屬性）, 1377
Attribute（*ast* 中的類F）, 2104
AttributeError, 111
attributes（*xml.dom.Node* 的屬性）, 1387
AttributesImpl（*xml.sax.xmlreader* 中的類F）, 1408
AttributesNSImpl（*xml.sax.xmlreader* 中的類F）, 1408
attribute（屬性）, **2229**
attroff()（*curses.window* 的方法）, 936
attron()（*curses.window* 的方法）, 936
attrset()（*curses.window* 的方法）, 936
audioop
    module, 2220
audit events（稽核事件）, 1855
audit()（於 *sys* 模組中）, 1920
auditing, 1920
AugAssign（*ast* 中的類F）, 2108
AUGUST（於 *calendar* 模組中）, 261
auth()（*ftplib.FTP_TLS* 的方法）, 1479
auth()（*smtplib.SMTP* 的方法）, 1494
authenticate()（*imaplib.IMAP4* 的方法）, 1484
AuthenticationError, 992
authenticators()（*netrc.netrc* 的方法）, 647
authkey（*multiprocessing.Process* 的屬性）, 991
auto（*enum* 中的類F）, 332
autocommit（*sqlite3.Connection* 的屬性）, 547
autorange()（*timeit.Timer* 的方法）, 1887
available_timezones()（於 *zoneinfo* 模組中）, 255
avoids_symlink_attacks（*shutil.rmtree* 的屬性）, 496
Await（*ast* 中的類F）, 2124
await_args（*unittest.mock.AsyncMock* 的屬性）, 1791
await_args_list（*unittest.mock.AsyncMock* 的屬性）, 1791
await_count（*unittest.mock.AsyncMock* 的屬性）, 1791
Awaitable（*collections.abc* 中的類F）, 284
Awaitable（*typing* 中的類F）, 1718
awaitable（可等待物件）, **2229**

B

-b
    compileall 命令列選項, 2148
    http.server 命令列選項, 1517
    unittest 命令列選項, 1749
b2a_base64()（於 *binascii* 模組中）, 1356
b2a_hex()（於 *binascii* 模組中）, 1357
b2a_qp()（於 *binascii* 模組中）, 1357
b2a_uu()（於 *binascii* 模組中）, 1356
b16decode()（於 *base64* 模組中）, 1354
b16encode()（於 *base64* 模組中）, 1354
b32decode()（於 *base64* 模組中）, 1353
b32encode()（於 *base64* 模組中）, 1353
b32hexdecode()（於 *base64* 模組中）, 1354
b32hexencode()（於 *base64* 模組中）, 1353
b64decode()（於 *base64* 模組中）, 1353
b64encode()（於 *base64* 模組中）, 1353
b85decode()（於 *base64* 模組中）, 1355
b85encode()（於 *base64* 模組中）, 1354
Babyl（*mailbox* 中的類F）, 1337
BabylMessage（*mailbox* 中的類F）, 1344
back()（於 *turtle* 模組中）, 1635
backend（於 *readline* 模組中）, 180
backslashreplace
    error handler's name（錯誤處理器名稱）, 197
backslashreplace_errors()（於 *codecs* 模組中）, 198
backup()（*sqlite3.Connection* 的方法）, 544
backward()（於 *turtle* 模組中）, 1635
BadGzipFile, 578
BadOptionError, 925

BadStatusLine, 1468
BadZipFile, 591
BadZipfile, 591
Barrier (*asyncio* 中的類🄵), 1109
Barrier (*multiprocessing* 中的類🄵), 999
Barrier (*threading* 中的類🄵), 981
Barrier() (*multiprocessing.managers.SyncManager* 的方法), 1006
base64
    encoding（編碼）, 1352
    module, 1352
    module（模組）, 1356
base_exec_prefix (於 *sys* 模組中), 1920
base_prefix (於 *sys* 模組中), 1920
BaseCGIHandler (*wsgiref.handlers* 中的類🄵), 1431
BaseCookie (*http.cookies* 中的類🄵), 1519
BaseException, 110
BaseExceptionGroup, 118
BaseHandler (*urllib.request* 中的類🄵), 1439
BaseHandler (*wsgiref.handlers* 中的類🄵), 1431
BaseHeader (*email.headerregistry* 中的類🄵), 1286
BaseHTTPRequestHandler (*http.server* 中的類🄵), 1512
BaseManager (*multiprocessing.managers* 中的類🄵), 1004
basename() (於 *os.path* 模組中), 469
BaseProtocol (*asyncio* 中的類🄵), 1151
BaseProxy (*multiprocessing.managers* 中的類🄵), 1009
BaseRequestHandler (*socketserver* 中的類🄵), 1507
BaseRotatingHandler (*logging.handlers* 中的類🄵), 792
BaseSelector (*selectors* 中的類🄵), 1245
BaseServer (*socketserver* 中的類🄵), 1505
BaseTransport (*asyncio* 中的類🄵), 1147
basicConfig() (於 *logging* 模組中), 776
BasicContext (於 *decimal* 模組中), 365
BasicInterpolation (*configparser* 中的類🄵), 631
batched() (於 *itertools* 模組中), 411
baudrate() (於 *curses* 模組中), 929
bbox() (*tkinter.ttk.Treeview* 的方法), 1609
BDADDR_ANY (於 *socket* 模組中), 1181
BDADDR_BREDR (於 *socket* 模組中), 1182
BDADDR_LE_PUBLIC (於 *socket* 模組中), 1182
BDADDR_LE_RANDOM (於 *socket* 模組中), 1182
BDADDR_LOCAL (於 *socket* 模組中), 1181
bdb
    module, 1859
    module（模組）, 1868
Bdb (*bdb* 中的類🄵), 1860
BdbQuit, 1859
BDFL, **2229**
beep() (於 *curses* 模組中), 929
Beep() (於 *winsound* 模組中), 2188
begin_fill() (於 *turtle* 模組中), 1645
begin_poly() (於 *turtle* 模組中), 1650
BEL (於 *curses.ascii* 模組中), 955
below() (*curses.panel.Panel* 的方法), 958
BELOW_NORMAL_PRIORITY_CLASS (於 *subprocess* 模組中), 1057
benchmarking（基准测試）, 751, 756
--best
    gzip 命令列選項, 581
betavariate() (於 *random* 模組中), 387
bgcolor() (於 *turtle* 模組中), 1652
bgpic() (於 *turtle* 模組中), 1652
bidirectional() (於 *unicodedata* 模組中), 177
bigaddrspacetest() (於 *test.support* 模組中), 1843
BigEndianStructure (*ctypes* 中的類🄵), 849
BigEndianUnion (*ctypes* 中的類🄵), 849
bigmemtest() (於 *test.support* 模組中), 1843
bin()
    built-in function, 9
Binary (*xmlrpc.client* 中的類🄵), 1534
binary file（二進位檔案）, **2229**
binary semaphores（二進位號🄵）, 1071
BINARY_OP (*opcode*), 2158
BINARY_SLICE (*opcode*), 2159
BinaryIO (*typing* 中的類🄵), 1705
binary（二進位）
    data（資料）, packing（打包）, 187
    literals（字面值）, 38
binascii
    module, 1356
--bind
    http.server 命令列選項, 1517
bind() (*inspect.Signature* 的方法), 2023
bind() (*socket.socket* 的方法), 1192
bind_partial() (*inspect.Signature* 的方法), 2023
bind_port() (於 *test.support.socket_helper* 模組中), 1846
bind_textdomain_codeset() (於 *locale* 模組中), 1577
bind_unix_socket() (於 *test.support.socket_helper* 模組中), 1846
bindtextdomain() (於 *gettext* 模組中), 1561
bindtextdomain() (於 *locale* 模組中), 1577
bind（元件）, 1589
binomialvariate() (於 *random* 模組中), 387
BinOp (*ast* 中的類🄵), 2102
bisect
    module, 290
bisect() (於 *bisect* 模組中), 291
bisect_left() (於 *bisect* 模組中), 290
bisect_right() (於 *bisect* 模組中), 291
bit_count() (*int* 的方法), 40
bit_length() (*int* 的方法), 40
bit_offset (*ctypes.CField* 的屬性), 852
bit_size (*ctypes.CField* 的屬性), 852
BitAnd (*ast* 中的類🄵), 2102
BitOr (*ast* 中的類🄵), 2102
bits_per_digit (*sys.int_info* 的屬性), 1934
bitwise（位元）
    operations（操作）, 40
BitXor (*ast* 中的類🄵), 2102
bk() (於 *turtle* 模組中), 1635

bkgd() (*curses.window* 的方法), 936
bkgdset() (*curses.window* 的方法), 936
blake2b() (於 *hashlib* 模組中), 655
blake2b, blake2s, 655
blake2b.MAX_DIGEST_SIZE (於 *hashlib* 模組中), 657
blake2b.MAX_KEY_SIZE (於 *hashlib* 模組中), 656
blake2b.PERSON_SIZE (於 *hashlib* 模組中), 656
blake2b.SALT_SIZE (於 *hashlib* 模組中), 656
blake2s() (於 *hashlib* 模組中), 655
blake2s.MAX_DIGEST_SIZE (於 *hashlib* 模組中), 657
blake2s.MAX_KEY_SIZE (於 *hashlib* 模組中), 656
blake2s.PERSON_SIZE (於 *hashlib* 模組中), 656
blake2s.SALT_SIZE (於 *hashlib* 模組中), 656
BLKTYPE (於 *tarfile* 模組中), 605
Blob (*sqlite3* 中的類🄵), 551
blobopen() (*sqlite3.Connection* 的方法), 538
block_on_close (*socketserver.ThreadingMixIn* 的屬性), 1504
block_size (*hmac.HMAC* 的屬性), 663
blocked_domains() (*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
BlockingIOError, 116, 736
blocksize (*http.client.HTTPConnection* 的屬性), 1470
body() (*tkinter.simpledialog.Dialog* 的方法), 1593
body_encode() (*email.charset.Charset* 的方法), 1314
body_encoding (*email.charset.Charset* 的屬性), 1314
body_line_iterator() (於 *email.iterators* 模組中), 1318
BOLD (於 *tkinter.font* 模組中), 1591
BOM (於 *codecs* 模組中), 196
BOM_BE (於 *codecs* 模組中), 196
BOM_LE (於 *codecs* 模組中), 196
BOM_UTF8 (於 *codecs* 模組中), 196
BOM_UTF16 (於 *codecs* 模組中), 196
BOM_UTF16_BE (於 *codecs* 模組中), 196
BOM_UTF16_LE (於 *codecs* 模組中), 196
BOM_UTF32 (於 *codecs* 模組中), 196
BOM_UTF32_BE (於 *codecs* 模組中), 196
BOM_UTF32_LE (於 *codecs* 模組中), 196
bool (🄵建類🄵), 9
BOOLEAN_STATES (*configparser.ConfigParser* 的屬性), 635
BooleanOptionalAction (*argparse* 中的類🄵), 873
Boolean（布林值）
    type（型🄵）, 9
Boolean（布林）
    object（物件）, 38
    operations（操作）, 37
    values, 45
BoolOp (*ast* 中的類🄵), 2102
bootstrap() (於 *ensurepip* 模組中), 1904
border() (*curses.window* 的方法), 937
borrowed reference（借用參照）, **2229**
bottom() (*curses.panel.Panel* 的方法), 958
bottom_panel() (於 *curses.panel* 模組中), 958
BoundArguments (*inspect* 中的類🄵), 2026
BoundaryError, 1285
BoundedSemaphore (*asyncio* 中的類🄵), 1109
BoundedSemaphore (*multiprocessing* 中的類🄵), 999
BoundedSemaphore (*threading* 中的類🄵), 979
BoundedSemaphore() (*multiprocessing.managers.SyncManager* 的方法), 1006
bounds() (*compression.zstd.CompressionParameter* 的方法), 569
bounds() (*compression.zstd.DecompressionParameter* 的方法), 572
box() (*curses.window* 的方法), 937
bpbynumber (*bdb.Breakpoint* 的屬性), 1860
bpformat() (*bdb.Breakpoint* 的方法), 1859
bplist (*bdb.Breakpoint* 的屬性), 1860
bpprint() (*bdb.Breakpoint* 的方法), 1860
BRANCH_LEFT (*monitoring event*), 1947
BRANCH_RIGHT (*monitoring event*), 1947
Break (*ast* 中的類🄵), 2111
break (*pdb command*), 1873
break_anywhere() (*bdb.Bdb* 的方法), 1862
break_here() (*bdb.Bdb* 的方法), 1862
break_long_words (*textwrap.TextWrapper* 的屬性), 176
break_on_hyphens (*textwrap.TextWrapper* 的屬性), 176
Breakpoint (*bdb* 中的類🄵), 1859
breakpoint()
    built-in function, 9
breakpointhook() (於 *sys* 模組中), 1921
breakpoints（中斷點）, 1621
broadcast_address (*ipaddress.IPv4Network* 的屬性), 1550
broadcast_address (*ipaddress.IPv6Network* 的屬性), 1553
broken (*asyncio.Barrier* 的屬性), 1110
broken (*threading.Barrier* 的屬性), 981
BrokenBarrierError, 981, 1110
BrokenExecutor, 1039
BrokenInterpreterPool, 1039
BrokenPipeError, 116
BrokenProcessPool, 1039
BrokenThreadPool, 1039
BROWSER, 1423, 1424
BS (於 *curses.ascii* 模組中), 955
BsdDbShelf (*shelve* 中的類🄵), 523
btlazy2 (*compression.zstd.Strategy* 的屬性), 572
btopt (*compression.zstd.Strategy* 的屬性), 573
BTPROTO_HCI (於 *socket* 模組中), 1181
BTPROTO_L2CAP (於 *socket* 模組中), 1181
BTPROTO_RFCOMM (於 *socket* 模組中), 1181
BTPROTO_SCO (於 *socket* 模組中), 1181
btultra (*compression.zstd.Strategy* 的屬性), 573
btultra2 (*compression.zstd.Strategy* 的屬性), 573
buf (*multiprocessing.shared_memory.SharedMemory* 的屬性), 1027
--buffer
    unittest 命令列選項, 1749

Buffer (*collections.abc* 中的類🄵), 285
buffer (*io.TextIOBase* 的屬性), 743
buffer (*unittest.TestResult* 的屬性), 1770
buffer protocol (緩衝區協定)
    binary sequence types (二進位序列型🄵), 66
    str (🄵建類🄵), 54
buffer size, I/O (緩衝區大小、I/O), 24
buffer_info() (*array.array* 的方法), 294
buffer_size (*xml.parsers.expat.xmlparser* 的屬性), 1414
buffer_text (*xml.parsers.expat.xmlparser* 的屬性), 1414
buffer_updated() (*asyncio.BufferedProtocol* 的方法), 1153
buffer_used (*xml.parsers.expat.xmlparser* 的屬性), 1414
BufferedIOBase (*io* 中的類🄵), 739
BufferedProtocol (*asyncio* 中的類🄵), 1152
BufferedRandom (*io* 中的類🄵), 743
BufferedReader (*io* 中的類🄵), 742
BufferedRWPair (*io* 中的類🄵), 743
BufferedWriter (*io* 中的類🄵), 742
BufferError, 110
BufferFlags (*inspect* 中的類🄵), 2034
BufferingFormatter (*logging* 中的類🄵), 769
BufferingHandler (*logging.handlers* 中的類🄵), 800
BufferTooShort, 992
BUILD_INTERPOLATION (*opcode*), 2163
BUILD_LIST (*opcode*), 2164
BUILD_MAP (*opcode*), 2164
build_opener() (於 *urllib.request* 模組中), 1436
BUILD_SET (*opcode*), 2164
BUILD_SLICE (*opcode*), 2169
BUILD_STRING (*opcode*), 2164
BUILD_TEMPLATE (*opcode*), 2163
BUILD_TUPLE (*opcode*), 2164
built-in function
    __import__(), 32
    abs(), 8
    aiter(), 8
    all(), 8
    anext(), 8
    any(), 9
    ascii(), 9
    bin(), 9
    breakpoint(), 9
    callable(), 10
    chr(), 10
    classmethod(), 10
    compile(), 11
    delattr(), 12
    dir(), 13
    divmod(), 13
    enumerate(), 13
    eval(), 14
    exec(), 15
    filter(), 15
    format(), 16
    getattr(), 17
    globals(), 17
    hasattr(), 17
    hash(), 17
    help(), 17
    hex(), 18
    id(), 18
    input(), 18
    isinstance(), 19
    issubclass(), 19
    iter(), 19
    len(), 20
    locals(), 20
    map(), 20
    max(), 21
    min(), 21
    multiprocessing.Manager(), 1004
    next(), 21
    oct(), 21
    open(), 22
    ord(), 24
    pow(), 24
    print(), 25
    property.deleter(), 26
    property.getter(), 26
    property.setter(), 26
    repr(), 26
    reversed(), 27
    round(), 27
    setattr(), 27
    sorted(), 28
    staticmethod(), 28
    sum(), 29
    vars(), 30
    zip(), 31
built-in function (🄵建函式)
    compile (編譯), 104, 305
    complex (🄵數), 38
    eval, 104, 312, 313
    exec, 15, 104
    float, 38
    hash (雜🄵), 48
    int, 38
    len, 46, 91
    max, 46
    min, 46
    slice (切片), 2169
    type (型🄵), 104
builtin_module_names (於 *sys* 模組中), 1920
BuiltinFunctionType (於 *types* 模組中), 306
BuiltinImporter (*importlib.machinery* 中的類🄵), 2069
BuiltinMethodType (於 *types* 模組中), 306
builtins
    module, 1957
builtins (🄵建)
    module (模組), 32

built-in（🄵建）
    type（型🄵）, 37
burst() (*imaplib.IMAP4.Idler* 的方法), 1486
busy_retry() (於 *test.support* 模組中), 1838
BUTTON_ALT (於 *curses* 模組中), 952
BUTTON_CTRL (於 *curses* 模組中), 952
BUTTON_SHIFT (於 *curses* 模組中), 952
buttonbox() (*tkinter.simpledialog.Dialog* 的方法), 1593
BUTTONn_CLICKED (於 *curses* 模組中), 952
BUTTONn_DOUBLE_CLICKED (於 *curses* 模組中), 952
BUTTONn_PRESSED (於 *curses* 模組中), 952
BUTTONn_RELEASED (於 *curses* 模組中), 952
BUTTONn_TRIPLE_CLICKED (於 *curses* 模組中), 952
bye() (於 *turtle* 模組中), 1658
byref() (於 *ctypes* 模組中), 842
byte_offset (*ctypes.CField* 的屬性), 852
byte_size (*ctypes.CField* 的屬性), 852
bytearray (🄵建類🄵), 68
bytearray（位元組陣列）
    formatting（格式化）, 80
    interpolation（插值）, 80
    methods（方法）, 69
    object（物件）, 48, 66, 68
Bytecode (*dis* 中的類🄵), 2152
BYTECODE_SUFFIXES (於 *importlib.machinery* 模組中), 2068
Bytecode.codeobj (於 *dis* 模組中), 2153
Bytecode.first_line (於 *dis* 模組中), 2153
BytecodeTestCase (*test.support.bytecode_helper* 中的類🄵), 1847
bytecode（位元組碼）, **2229**
byte-code（位元組碼）
    file（檔案）, 2145
byteorder (於 *sys* 模組中), 1920
bytes (*uuid.UUID* 的屬性), 1498
bytes (🄵建類🄵), 67
bytes-like object（類位元組串物件）, **2229**
bytes_le (*uuid.UUID* 的屬性), 1498
bytes_warning (*sys.flags* 的屬性), 1926
BytesFeedParser (*email.parser* 中的類🄵), 1272
BytesGenerator (*email.generator* 中的類🄵), 1275
BytesHeaderParser (*email.parser* 中的類🄵), 1274
BytesIO (*io* 中的類🄵), 741
BytesParser (*email.parser* 中的類🄵), 1273
ByteString (*collections.abc* 中的類🄵), 283
ByteString (*typing* 中的類🄵), 1716
byteswap() (*array.array* 的方法), 295
BytesWarning, 118
bytes（位元組）
    formatting（格式化）, 80
    interpolation（插值）, 80
    methods（方法）, 69
    object（物件）, 66, 67
    str（🄵建類🄵）, 54
bz2
    module, 581
BZ2Compressor (*bz2* 中的類🄵), 583
BZ2Decompressor (*bz2* 中的類🄵), 583
BZ2File (*bz2* 中的類🄵), 581

C

C
    language（語言）, 38
    structures（結構）, 187
-C
    dis 命令列選項, 2152
    trace 命令列選項, 1891
    uuid 命令列選項, 1501
-c
    calendar 命令列選項, 263
    idle 命令列選項, 1624
    pdb 命令列選項, 1869
    random 命令列選項, 392
    tarfile 命令列選項, 616
    trace 命令列選項, 1891
    unittest 命令列選項, 1749
    zipapp 命令列選項, 1915
    zipfile 命令列選項, 601
C14NWriterTarget (*xml.etree.ElementTree* 中的類🄵), 1382
c_bool (*ctypes* 中的類🄵), 849
c_byte (*ctypes* 中的類🄵), 847
c_char (*ctypes* 中的類🄵), 847
c_char_p (*ctypes* 中的類🄵), 847
C_CONTIGUOUS (*inspect.BufferFlags* 的屬性), 2034
c_contiguous (*memoryview* 的屬性), 89
c_double (*ctypes* 中的類🄵), 847
c_double_complex (*ctypes* 中的類🄵), 847
c_float (*ctypes* 中的類🄵), 847
c_float_complex (*ctypes* 中的類🄵), 847
c_int (*ctypes* 中的類🄵), 847
c_int8 (*ctypes* 中的類🄵), 847
c_int16 (*ctypes* 中的類🄵), 848
c_int32 (*ctypes* 中的類🄵), 848
c_int64 (*ctypes* 中的類🄵), 848
c_long (*ctypes* 中的類🄵), 848
c_longdouble (*ctypes* 中的類🄵), 847
c_longdouble_complex (*ctypes* 中的類🄵), 847
c_longlong (*ctypes* 中的類🄵), 848
C_RAISE (*monitoring event*), 1947
C_RETURN (*monitoring event*), 1947
c_short (*ctypes* 中的類🄵), 848
c_size_t (*ctypes* 中的類🄵), 848
c_ssize_t (*ctypes* 中的類🄵), 848
c_time_t (*ctypes* 中的類🄵), 848
c_ubyte (*ctypes* 中的類🄵), 848
c_uint (*ctypes* 中的類🄵), 848
c_uint8 (*ctypes* 中的類🄵), 848
c_uint16 (*ctypes* 中的類🄵), 848
c_uint32 (*ctypes* 中的類🄵), 848
c_uint64 (*ctypes* 中的類🄵), 848
c_ulong (*ctypes* 中的類🄵), 848
c_ulonglong (*ctypes* 中的類🄵), 848
c_ushort (*ctypes* 中的類🄵), 849
c_void_p (*ctypes* 中的類🄵), 849

c_wchar (*ctypes* 中的類🄵), 849
c_wchar_p (*ctypes* 中的類🄵), 849
CACHE (*opcode*), 2158
cache() (於 *functools* 模組中), 425
cache_clear() (*functools.lru_cache* 的方法), 427
cache_from_source() (於 *importlib.util* 模組中), 2074
cache_info() (*functools.lru_cache* 的方法), 427
cached (*importlib.machinery.ModuleSpec* 的屬性), 2072
cached_property() (於 *functools* 模組中), 425
CacheFTPHandler (*urllib.request* 中的類🄵), 1440
calcobjsize() (於 *test.support* 模組中), 1841
calcsize() (於 *struct* 模組中), 188
calcvobjsize() (於 *test.support* 模組中), 1841
calendar
    module, 255
Calendar (*calendar* 中的類🄵), 256
calendar 命令列選項
    -c, 263
    --css, 263
    -e, 262
    --encoding, 262
    -f, 263
    --first-weekday, 263
    -h, 262
    --help, 262
    -L, 262
    -l, 263
    --lines, 263
    --locale, 262
    -m, 263
    month, 263
    --months, 263
    -s, 263
    --spacing, 263
    -t, 263
    --type, 263
    -w, 263
    --width, 263
    year, 263
calendar() (於 *calendar* 模組中), 260
Call (*ast* 中的類🄵), 2103
CALL (*monitoring event*), 1947
CALL (*opcode*), 2168
call() (*concurrent.interpreters.Interpreter* 的方法), 1042
call() (於 *operator* 模組中), 438
call() (於 *subprocess* 模組中), 1057
call() (於 *unittest.mock* 模組中), 1807
call_annotate_function() (於 *annotationlib* 模組中), 2038
call_args (*unittest.mock.Mock* 的屬性), 1784
call_args_list (*unittest.mock.Mock* 的屬性), 1785
call_at() (*asyncio.loop* 的方法), 1124
call_count (*unittest.mock.Mock* 的屬性), 1783
call_evaluate_function() (於 *annotationlib* 模組中), 2038
call_exception_handler() (*asyncio.loop* 的方法), 1136
CALL_FUNCTION_EX (*opcode*), 2169
call_in_thread() (*concurrent.interpreters.Interpreter* 的方法), 1042
CALL_INTRINSIC_1 (*opcode*), 2171
CALL_INTRINSIC_2 (*opcode*), 2171
CALL_KW (*opcode*), 2168
call_later() (*asyncio.loop* 的方法), 1124
call_list() (*unittest.mock.call* 的方法), 1808
call_soon() (*asyncio.loop* 的方法), 1123
call_soon_threadsafe() (*asyncio.loop* 的方法), 1123
call_tracing() (於 *sys* 模組中), 1921
Callable (*collections.abc* 中的類🄵), 283
Callable (於 *typing* 模組中), 1718
callable()
    built-in function, 10
CallableProxyType (於 *weakref* 模組中), 300
callable（可呼叫物件）, **2229**
callback (*optparse.Option* 的屬性), 912
callback() (*contextlib.ExitStack* 的方法), 1988
callback_args (*optparse.Option* 的屬性), 912
callback_kwargs (*optparse.Option* 的屬性), 912
callbacks (於 *gc* 模組中), 2015
callback（回呼）, **2229**
called (*unittest.mock.Mock* 的屬性), 1783
CalledProcessError, 1046
CAN (於 *curses.ascii* 模組中), 956
CAN_BCM (於 *socket* 模組中), 1179
can_change_color() (於 *curses* 模組中), 929
can_fetch() (*urllib.robotparser.RobotFileParser* 的方法), 1462
CAN_ISOTP (於 *socket* 模組中), 1180
CAN_J1939 (於 *socket* 模組中), 1180
CAN_RAW_FD_FRAMES (於 *socket* 模組中), 1180
CAN_RAW_JOIN_FILTERS (於 *socket* 模組中), 1180
can_symlink() (於 *test.support.os_helper* 模組中), 1850
can_write_eof() (*asyncio.StreamWriter* 的方法), 1102
can_write_eof() (*asyncio.WriteTransport* 的方法), 1149
can_xattr() (於 *test.support.os_helper* 模組中), 1850
CANCEL (於 *tkinter.messagebox* 模組中), 1597
cancel() (*asyncio.Future* 的方法), 1145
cancel() (*asyncio.Handle* 的方法), 1138
cancel() (*asyncio.Task* 的方法), 1096
cancel() (*concurrent.futures.Future* 的方法), 1037
cancel() (*sched.scheduler* 的方法), 1064
cancel() (*threading.Timer* 的方法), 980
cancel() (*tkinter.dnd.DndHandler* 的方法), 1598
cancel_command() (*tkinter.filedialog.FileDialog* 的方法), 1594
cancel_dump_traceback_later() (於 *faulthandler* 模組中), 1866
cancel_join_thread() (*multiprocessing.Queue* 的方法), 994

cancelled() (*asyncio.Future* 的方法), 1145
cancelled() (*asyncio.Handle* 的方法), 1139
cancelled() (*asyncio.Task* 的方法), 1097
cancelled() (*concurrent.futures.Future* 的方法), 1037
CancelledError, 1039, 1118
cancelling() (*asyncio.Task* 的方法), 1098
CannotSendHeader, 1468
CannotSendRequest, 1468
canonic() (*bdb.Bdb* 的方法), 1860
canonical() (*decimal.Context* 的方法), 368
canonical() (*decimal.Decimal* 的方法), 359
canonicalize() (於 *xml.etree.ElementTree* 模組中), 1372
capa() (*poplib.POP3* 的方法), 1481
capitalize() (*bytearray* 的方法), 75
capitalize() (*bytes* 的方法), 75
capitalize() (*str* 的方法), 54
capitals (*decimal.Context* 的屬性), 366
CapsuleType (*types* 中的類🄵), 308
capture_call_graph() (於 *asyncio* 模組中), 1120
captured_stderr() (於 *test.support* 模組中), 1840
captured_stdin() (於 *test.support* 模組中), 1840
captured_stdout() (於 *test.support* 模組中), 1840
captureWarnings() (於 *logging* 模組中), 778
capwords() (於 *string* 模組中), 134
casefold() (*str* 的方法), 54
cast() (*memoryview* 的方法), 86
cast() (於 *ctypes* 模組中), 842
cast() (於 *typing* 模組中), 1706
--catch
    unittest 命令列選項, 1749
catch_threading_exception() (於 *test.support.threading_helper* 模組中), 1848
catch_unraisable_exception() (於 *test.support* 模組中), 1843
catch_warnings (*warnings* 中的類🄵), 1968
category() (於 *unicodedata* 模組中), 177
cbreak() (於 *curses* 模組中), 929
cbrt() (於 *math* 模組中), 345
ccc() (*ftplib.FTP_TLS* 的方法), 1479
C-contiguous（C 連續的）, 2230
cdf() (*statistics.NormalDist* 的方法), 404
CDLL (*ctypes* 中的類🄵), 836
ceil() (於 *math* 模組中), 342
ceil()（於 math 模組）, 39
CellType (於 *types* 模組中), 305
center() (*bytearray* 的方法), 73
center() (*bytes* 的方法), 73
center() (*str* 的方法), 54
CERT_NONE (於 *ssl* 模組中), 1207
CERT_OPTIONAL (於 *ssl* 模組中), 1208
CERT_REQUIRED (於 *ssl* 模組中), 1208
cert_store_stats() (*ssl.SSLContext* 的方法), 1219
cert_time_to_seconds() (於 *ssl* 模組中), 1206
CertificateError, 1206
certificates（憑證）, 1227
CField (*ctypes* 中的類🄵), 851
cfmakecbreak() (於 *tty* 模組中), 2200
cfmakeraw() (於 *tty* 模組中), 2200
CFUNCTYPE() (於 *ctypes* 模組中), 839
cget() (*tkinter.font.Font* 的方法), 1592
cgi
    module, 2220
--cgi
    http.server 命令列選項, 1517
cgi_directories (*http.server.CGIHTTPRequestHandler* 的屬性), 1516
CGIHandler (*wsgiref.handlers* 中的類🄵), 1430
CGIHTTPRequestHandler (*http.server* 中的類🄵), 1516
cgitb
    module, 2220
CGIXMLRPCRequestHandler (*xmlrpc.server* 中的類🄵), 1538
chain() (於 *itertools* 模組中), 412
chain_log (*compression.zstd.CompressionParameter* 的屬性), 570
chaining
    exception（例外）, 109
chaining（鏈結）
    comparisons（比較）, 38
ChainMap (*collections* 中的類🄵), 264
ChainMap (*typing* 中的類🄵), 1715
change_cwd() (於 *test.support.os_helper* 模組中), 1850
CHANNEL_BINDING_TYPES (於 *ssl* 模組中), 1213
CHAR_MAX (於 *locale* 模組中), 1575
CharacterDataHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
characters() (*xml.sax.handler.ContentHandler* 的方法), 1405
characters_written (*BlockingIOError* 的屬性), 116
character（字元）, 176
charmap_build() (於 *codecs* 模組中), 194
Charset (*email.charset* 中的類🄵), 1313
charset() (*gettext.NullTranslations* 的方法), 1564
chdir() (於 *contextlib* 模組中), 1985
chdir() (於 *os* 模組中), 691
check (*lzma.LZMADecompressor* 的屬性), 588
check() (*imaplib.IMAP4* 的方法), 1485
check() (於 *tabnanny* 模組中), 2143
check__all__() (於 *test.support* 模組中), 1844
check_call() (於 *subprocess* 模組中), 1058
check_disallow_instantiation() (於 *test.support* 模組中), 1845
CHECK_EG_MATCH (*opcode*), 2161
CHECK_EXC_MATCH (*opcode*), 2161
check_free_after_iterating() (於 *test.support* 模組中), 1844
check_hostname (*ssl.SSLContext* 的屬性), 1224
check_impl_detail() (於 *test.support* 模組中), 1840

check_no_resource_warning() (於 test.support.warnings_helper 模組中), 1852
check_output() (doctest.OutputChecker 的方法), 1742
check_output() (於 subprocess 模組中), 1058
check_returncode() (subprocess.CompletedProcess 的方法), 1045
check_syntax_error() (於 test.support 模組中), 1843
check_syntax_warning() (於 test.support.warnings_helper 模組中), 1852
check_unused_args() (string.Formatter 的方法), 125
check_warnings() (於 test.support.warnings_helper 模組中), 1852
checkcache() (於 linecache 模組中), 492
CHECKED_HASH (py_compile.PycInvalidationMode 的屬性), 2146
checkfuncname() (於 bdb 模組中), 1864
checksizeof() (於 test.support 模組中), 1841
checksum_flag (compression.zstd.CompressionParameter 的屬性), 571
checksum (核對和)
    Cyclic Redundancy Check (循環冗余核對), 575
chflags() (於 os 模組中), 691
chgat() (curses.window 的方法), 937
childNodes (xml.dom.Node 的屬性), 1387
ChildProcessError, 116
children (pyclbr.Class 的屬性), 2145
children (pyclbr.Function 的屬性), 2144
children (tkinter.Tk 的屬性), 1581
chksum (tarfile.TarInfo 的屬性), 611
chmod() (pathlib.Path 的方法), 465
chmod() (於 os 模組中), 692
--choice
    random 命令列選項, 392
choice() (於 random 模組中), 386
choice() (於 secrets 模組中), 664
choices (optparse.Option 的屬性), 912
choices() (於 random 模組中), 386
Chooser (tkinter.colorchooser 中的類🄵), 1591
chown() (於 os 模組中), 693
chown() (於 shutil 模組中), 497
chr()
    built-in function, 10
chroot() (於 os 模組中), 693
CHRTYPE (於 tarfile 模組中), 605
chunk
    module, 2220
cipher() (ssl.SSLSocket 的方法), 1216
circle() (於 turtle 模組中), 1638
CIRCUMFLEX (於 token 模組中), 2137
CIRCUMFLEXEQUAL (於 token 模組中), 2138
clamp (decimal.Context 的屬性), 366
Clamped (decimal 中的類🄵), 372
Class (pyclbr 中的類🄵), 2145
Class (symtable 中的類🄵), 2131
CLASS (symtable.SymbolTableType 的屬性), 2130
class variable (類🄵變數), **2230**
ClassDef (ast 中的類🄵), 2123
classmethod()
    built-in function, 10
ClassMethodDescriptorType (於 types 模組中), 306
ClassVar (於 typing 模組中), 1684
class (類🄵), **2229**
CLD_CONTINUED (於 os 模組中), 728
CLD_DUMPED (於 os 模組中), 728
CLD_EXITED (於 os 模組中), 728
CLD_KILLED (於 os 模組中), 728
CLD_STOPPED (於 os 模組中), 728
CLD_TRAPPED (於 os 模組中), 728
clean() (mailbox.Maildir 的方法), 1333
cleandoc() (於 inspect 模組中), 2022
CleanImport (test.support.import_helper 中的類🄵), 1852
cleanup() (tempfile.TemporaryDirectory 的方法), 485
CLEANUP_THROW (opcode), 2160
--clear
    venv 命令列選項, 1906
clear (pdb command), 1873
Clear Breakpoint, 1621
clear() (array.array 的方法), 295
clear() (asyncio.Event 的方法), 1107
clear() (collections.deque 的方法), 269
clear() (curses.window 的方法), 937
clear() (dbm.gnu.gdbm 的方法), 529
clear() (dbm.ndbm.ndbm 的方法), 530
clear() (dict 的方法), 93
clear() (email.message.EmailMessage 的方法), 1271
clear() (frozenset 的方法), 91
clear() (http.cookiejar.CookieJar 的方法), 1524
clear() (mailbox.Mailbox 的方法), 1331
clear() (sequence 的方法), 48
clear() (threading.Event 的方法), 979
clear() (於 turtle 模組中), 1646
clear() (xml.etree.ElementTree.Element 的方法), 1377
clear_all_breaks() (bdb.Bdb 的方法), 1863
clear_all_file_breaks() (bdb.Bdb 的方法), 1863
clear_bpbynumber() (bdb.Bdb 的方法), 1863
clear_break() (bdb.Bdb 的方法), 1863
clear_cache() (於 filecmp 模組中), 481
clear_cache() (zoneinfo.ZoneInfo 的類🄵方法), 253
clear_content() (email.message.EmailMessage 的方法), 1271
clear_flags() (decimal.Context 的方法), 367
clear_frames() (於 traceback 模組中), 2002
clear_history() (於 readline 模組中), 182
clear_memo() (pickle.Pickler 的方法), 509
clear_overloads() (於 typing 模組中), 1709
clear_session_cookies() (http.cookiejar.CookieJar 的方法), 1524
clear_tool_id() (於 sys.monitoring 模組中), 1947
clear_traces() (於 tracemalloc 模組中), 1897

clear_traps() (*decimal.Context* 的方法), 367
clearcache() (於 *linecache* 模組中), 492
clearok() (*curses.window* 的方法), 937
clearscreen() (於 *turtle* 模組中), 1652
clearstamp() (於 *turtle* 模組中), 1639
clearstamps() (於 *turtle* 模組中), 1639
Client() (於 *multiprocessing.connection* 模組中), 1013
client_address (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
client_address (*socketserver.BaseRequestHandler* 的屬性), 1507
CLOCK_BOOTTIME (於 *time* 模組中), 757
clock_getres() (於 *time* 模組中), 749
clock_gettime() (於 *time* 模組中), 749
clock_gettime_ns() (於 *time* 模組中), 749
CLOCK_HIGHRES (於 *time* 模組中), 757
CLOCK_MONOTONIC (於 *time* 模組中), 757
CLOCK_MONOTONIC_RAW (於 *time* 模組中), 757
CLOCK_MONOTONIC_RAW_APPROX (於 *time* 模組中), 758
CLOCK_PROCESS_CPUTIME_ID (於 *time* 模組中), 758
CLOCK_PROF (於 *time* 模組中), 758
CLOCK_REALTIME (於 *time* 模組中), 758
clock_seq (*uuid.UUID* 的屬性), 1498
clock_seq_hi_variant (*uuid.UUID* 的屬性), 1498
clock_seq_low (*uuid.UUID* 的屬性), 1498
clock_settime() (於 *time* 模組中), 749
clock_settime_ns() (於 *time* 模組中), 749
CLOCK_TAI (於 *time* 模組中), 758
CLOCK_THREAD_CPUTIME_ID (於 *time* 模組中), 758
CLOCK_UPTIME (於 *time* 模組中), 758
CLOCK_UPTIME_RAW (於 *time* 模組中), 758
CLOCK_UPTIME_RAW_APPROX (於 *time* 模組中), 758
clone() (*email.generator.BytesGenerator* 的方法), 1276
clone() (*email.generator.Generator* 的方法), 1277
clone() (*email.policy.Policy* 的方法), 1280
clone() (於 *turtle* 模組中), 1651
CLONE_FILES (於 *os* 模組中), 676
CLONE_FS (於 *os* 模組中), 676
CLONE_NEWCGROUP (於 *os* 模組中), 676
CLONE_NEWIPC (於 *os* 模組中), 676
CLONE_NEWNET (於 *os* 模組中), 676
CLONE_NEWNS (於 *os* 模組中), 676
CLONE_NEWPID (於 *os* 模組中), 676
CLONE_NEWTIME (於 *os* 模組中), 676
CLONE_NEWUSER (於 *os* 模組中), 676
CLONE_NEWUTS (於 *os* 模組中), 676
CLONE_SIGHAND (於 *os* 模組中), 676
CLONE_SYSVSEM (於 *os* 模組中), 676
CLONE_THREAD (於 *os* 模組中), 676
CLONE_VM (於 *os* 模組中), 676
cloneNode() (*xml.dom.Node* 的方法), 1388
close() (*asyncio.BaseTransport* 的方法), 1148
close() (*asyncio.loop* 的方法), 1122
close() (*asyncio.Runner* 的方法), 1078
close() (*asyncio.Server* 的方法), 1139
close() (*asyncio.StreamWriter* 的方法), 1101
close() (*asyncio.SubprocessTransport* 的方法), 1151
close() (*concurrent.interpreters.Interpreter* 的方法), 1042
close() (*contextlib.ExitStack* 的方法), 1988
close() (*dbm.dumb.dumbdbm* 的方法), 531
close() (*dbm.gnu.gdbm* 的方法), 529
close() (*dbm.ndbm.ndbm* 的方法), 530
close() (*email.parser.BytesFeedParser* 的方法), 1273
close() (*ftplib.FTP* 的方法), 1478
close() (*html.parser.HTMLParser* 的方法), 1361
close() (*http.client.HTTPConnection* 的方法), 1470
close() (*imaplib.IMAP4* 的方法), 1485
close() (*io.IOBase* 的方法), 737
close() (*logging.FileHandler* 的方法), 791
close() (*logging.Handler* 的方法), 767
close() (*logging.handlers.MemoryHandler* 的方法), 801
close() (*logging.handlers.NTEventLogHandler* 的方法), 799
close() (*logging.handlers.SocketHandler* 的方法), 796
close() (*logging.handlers.SysLogHandler* 的方法), 798
close() (*mailbox.Mailbox* 的方法), 1332
close() (*mailbox.Maildir* 的方法), 1335
close() (*mailbox.MH* 的方法), 1337
close() (*mmap.mmap* 的方法), 1258
close() (*multiprocessing.connection.Connection* 的方法), 998
close() (*multiprocessing.connection.Listener* 的方法), 1014
close() (*multiprocessing.pool.Pool* 的方法), 1012
close() (*multiprocessing.Process* 的方法), 991
close() (*multiprocessing.Queue* 的方法), 994
close() (*multiprocessing.shared_memory.SharedMemory* 的方法), 1027
close() (*multiprocessing.SimpleQueue* 的方法), 995
close() (*os.scandir* 的方法), 700
close() (*select.devpoll* 的方法), 1239
close() (*select.epoll* 的方法), 1240
close() (*select.kqueue* 的方法), 1241
close() (*selectors.BaseSelector* 的方法), 1246
close() (*shelve.Shelf* 的方法), 522
close() (*socket.socket* 的方法), 1192
close() (*sqlite3.Blob* 的方法), 551
close() (*sqlite3.Connection* 的方法), 538
close() (*sqlite3.Cursor* 的方法), 549
close() (*tarfile.TarFile* 的方法), 609
close() (*urllib.request.BaseHandler* 的方法), 1443
close() (*wave.Wave_read* 的方法), 1558
close() (*wave.Wave_write* 的方法), 1559
close() (於 *fileinput* 模組中), 927
close() (於 *os* 模組中), 677
close() (於 *socket* 模組中), 1186
Close() (*winreg.PyHKEY* 的方法), 2188
close() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381

close() (*xml.etree.ElementTree.XMLParser* 的方法), 1382
close() (*xml.etree.ElementTree.XMLPullParser* 的方法), 1383
close() (*xml.sax.xmlreader.IncrementalParser* 的方法), 1409
close() (*zipfile.ZipFile* 的方法), 594
close_clients() (*asyncio.Server* 的方法), 1139
close_connection (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
CloseBoundaryNotFoundDefect, 1285
closed (*http.client.HTTPResponse* 的屬性), 1472
closed (*io.IOBase* 的屬性), 737
closed (*mmap.mmap* 的屬性), 1258
closed (*select.devpoll* 的屬性), 1239
closed (*select.epoll* 的屬性), 1240
closed (*select.kqueue* 的屬性), 1241
CloseKey() (於 *winreg* 模組中), 2180
closelog() (於 *syslog* 模組中), 2210
closerange() (於 *os* 模組中), 677
closing() (於 *contextlib* 模組中), 1982
closure variable（閉包變數）, **2230**
clrtobot() (*curses.window* 的方法), 937
clrtoeol() (*curses.window* 的方法), 937
cmath
    module, 349
cmd
    module, 959
    module（模組）, 1868
Cmd (*cmd* 中的類🄵), 959
cmd (*subprocess.CalledProcessError* 的屬性), 1046
cmd (*subprocess.TimeoutExpired* 的屬性), 1046
cmdloop() (*cmd.Cmd* 的方法), 960
cmdqueue (*cmd.Cmd* 的屬性), 961
cmp() (於 *filecmp* 模組中), 481
cmp_op (於 *dis* 模組中), 2173
cmp_to_key() (於 *functools* 模組中), 426
cmpfiles() (於 *filecmp* 模組中), 481
CMSG_LEN() (於 *socket* 模組中), 1189
CMSG_SPACE() (於 *socket* 模組中), 1190
CO_ASYNC_GENERATOR (於 *inspect* 模組中), 2033
CO_COROUTINE (於 *inspect* 模組中), 2033
CO_GENERATOR (於 *inspect* 模組中), 2033
CO_HAS_DOCSTRING (於 *inspect* 模組中), 2034
CO_ITERABLE_COROUTINE (於 *inspect* 模組中), 2033
CO_METHOD (於 *inspect* 模組中), 2034
CO_NESTED (於 *inspect* 模組中), 2033
CO_NEWLOCALS (於 *inspect* 模組中), 2033
CO_OPTIMIZED (於 *inspect* 模組中), 2033
CO_VARARGS (於 *inspect* 模組中), 2033
CO_VARKEYWORDS (於 *inspect* 模組中), 2033
code
    module, 2047
code (*SystemExit* 的屬性), 115
code (*urllib.error.HTTPError* 的屬性), 1461
code (*urllib.response.addinfourl* 的屬性), 1452
code (*xml.etree.ElementTree.ParseError* 的屬性), 1384
code (*xml.parsers.expat.ExpatError* 的屬性), 1416
code object（程式碼物件）, 103, 524
code_context (*inspect.FrameInfo* 的屬性), 2029
code_context (*inspect.Traceback* 的屬性), 2029
code_info() (於 *dis* 模組中), 2153
Codec (*codecs* 中的類🄵), 199
CodecInfo (*codecs* 中的類🄵), 194
CodecRegistryError, 210
Codecs, 194
    decode（解碼）, 194
    encode（編碼）, 194
codecs
    module, 194
coded_value (*http.cookies.Morsel* 的屬性), 1520
codeop
    module, 2049
codepoint2name (於 *html.entities* 模組中), 1364
codes (於 *xml.parsers.expat.errors* 模組中), 1418
CODESET (於 *locale* 模組中), 1571
CodeType (*types* 中的類🄵), 305
col_offset (*ast.AST* 的屬性), 2096
collapse_addresses() (於 *ipaddress* 模組中), 1556
collapse_rfc2231_value() (於 *email.utils* 模組中), 1318
collect() (於 *gc* 模組中), 2012
collectedDurations (*unittest.TestResult* 的屬性), 1770
Collection (*collections.abc* 中的類🄵), 283
Collection (*typing* 中的類🄵), 1716
collections
    module, 263
collections.abc
    module, 280
colno (*json.JSONDecodeError* 的屬性), 1326
colno (*re.PatternError* 的屬性), 151
colno (*tomllib.TOMLDecodeError* 的屬性), 645
colno (*traceback.FrameSummary* 的屬性), 2006
colon (*mailbox.Maildir* 的屬性), 1333
COLON (於 *token* 模組中), 2136
COLONEQUAL (於 *token* 模組中), 2138
color() (於 *turtle* 模組中), 1644
COLOR_BLACK (於 *curses* 模組中), 953
COLOR_BLUE (於 *curses* 模組中), 953
color_content() (於 *curses* 模組中), 929
COLOR_CYAN (於 *curses* 模組中), 953
COLOR_GREEN (於 *curses* 模組中), 953
COLOR_MAGENTA (於 *curses* 模組中), 953
color_pair() (於 *curses* 模組中), 929
COLOR_PAIRS (於 *curses* 模組中), 942
COLOR_RED (於 *curses* 模組中), 953
COLOR_WHITE (於 *curses* 模組中), 953
COLOR_YELLOW (於 *curses* 模組中), 953
colormode() (於 *turtle* 模組中), 1657
COLORS (於 *curses* 模組中), 942
colorsys
    module, 1560
COLS (於 *curses* 模組中), 942
column() (*tkinter.ttk.Treeview* 的方法), 1609
columnize() (*cmd.Cmd* 的方法), 960

COLUMNS, 935
columns (*os.terminal_size* 的屬性), 689
comb() (於 *math* 模組中), 341
combinations() (於 *itertools* 模組中), 412
combinations_with_replacement() (於 *itertools* 模組中), 413
combine() (*datetime.datetime* 的類Ⓕ方法), 226
combining() (於 *unicodedata* 模組中), 178
Combobox (*tkinter.ttk* 中的類Ⓕ), 1602
COMError, 853
COMMA (於 *token* 模組中), 2136
--command
    pdb 命令列選項, 1869
command (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
CommandCompiler (*codeop* 中的類Ⓕ), 2049
commands (*pdb command*), 1873
comment (*http.cookiejar.Cookie* 的屬性), 1529
comment (*http.cookies.Morsel* 的屬性), 1520
COMMENT (於 *token* 模組中), 2134
comment (*zipfile.ZipFile* 的屬性), 597
comment (*zipfile.ZipInfo* 的屬性), 600
Comment() (於 *xml.etree.ElementTree* 模組中), 1373
comment() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
comment() (*xml.sax.handler.LexicalHandler* 的方法), 1406
comment_url (*http.cookiejar.Cookie* 的屬性), 1529
commenters (*shlex.shlex* 的屬性), 2193
CommentHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
commit() (*sqlite3.Connection* 的方法), 538
common (*filecmp.dircmp* 的屬性), 482
Common Vulnerabilities and Exposures
    CVE 2020-10735, 105
    CVE 2023-52425, 1365
Common Weakness Enumeration
    CWE 257, 665
common_dirs (*filecmp.dircmp* 的屬性), 482
common_files (*filecmp.dircmp* 的屬性), 482
common_funny (*filecmp.dircmp* 的屬性), 482
common_types (於 *mimetypes* 模組中), 1349
commonpath() (於 *os.path* 模組中), 469
commonprefix() (於 *os.path* 模組中), 469
communicate() (*asyncio.subprocess.Process* 的方法), 1113
communicate() (*subprocess.Popen* 的方法), 1053
--compact
    json 命令列選項, 1329
Compare (*ast* 中的類Ⓕ), 2103
compare() (*decimal.Context* 的方法), 368
compare() (*decimal.Decimal* 的方法), 359
compare() (*difflib.Differ* 的方法), 168
compare() (於 *ast* 模組中), 2128
compare_digest() (於 *hmac* 模組中), 663
compare_digest() (於 *secrets* 模組中), 665
compare_networks() (*ipaddress.IPv4Network* 的方法), 1552
compare_networks() (*ipaddress.IPv6Network* 的方法), 1553
COMPARE_OP (*opcode*), 2165
compare_signal() (*decimal.Context* 的方法), 368
compare_signal() (*decimal.Decimal* 的方法), 359
compare_to() (*tracemalloc.Snapshot* 的方法), 1900
compare_total() (*decimal.Context* 的方法), 368
compare_total() (*decimal.Decimal* 的方法), 359
compare_total_mag() (*decimal.Context* 的方法), 368
compare_total_mag() (*decimal.Decimal* 的方法), 359
comparing (比較)
    objects (物件), 38
COMPARISON_FLAGS (於 *doctest* 模組中), 1732
comparisons (比較)
    chaining (鏈結), 38
comparison (比較)
    operator (運算子), 38
Compat32 (*email.policy* 中的類Ⓕ), 1284
compat32 (於 *email.policy* 模組中), 1284
Compile (*codeop* 中的類Ⓕ), 2049
compile()
    built-in function, 11
compile() (於 *py_compile* 模組中), 2145
compile() (於 *re* 模組中), 148
compile_command() (於 *code* 模組中), 2048
compile_command() (於 *codeop* 模組中), 2049
compile_dir() (於 *compileall* 模組中), 2149
compile_file() (於 *compileall* 模組中), 2150
compile_path() (於 *compileall* 模組中), 2150
compileall
    module, 2147
compileall 命令列選項
    -b, 2148
    -d, 2147
    directory, 2147
    -e, 2148
    -f, 2147
    file, 2147
    --hardlink-dupes, 2148
    -i, 2148
    --invalidation-mode, 2148
    -j, 2148
    -l, 2147
    -o, 2148
    -p, 2148
    -q, 2147
    -r, 2148
    -s, 2148
    -x, 2148
compiler_flag (*__future__._Feature* 的屬性), 2012
compile (編譯)
    built-in function (Ⓕ建函式), 104, 305
complete() (*rlcompleter.Completer* 的方法), 185
complete_statement() (於 *sqlite3* 模組中), 535
completedefault() (*cmd.Cmd* 的方法), 960
CompletedProcess (*subprocess* 中的類Ⓕ), 1045

Completer (*rlcompleter* 中的類🄵), 185
Complex (*numbers* 中的類🄵), 337
complex (🄵建類🄵), 11
complex number (🄵數), **2230**
    literals (字面值), 38
    object (物件), 38
complex (🄵數)
    built-in function (🄵建函式), 38
comprehension (*ast* 中的類🄵), 2105
--compress
    zipapp 命令列選項, 1915
compress() (*bz2.BZ2Compressor* 的方法), 583
compress() (*compression.zstd.ZstdCompressor* 的方法), 566
compress() (*lzma.LZMACompressor* 的方法), 588
compress() (於 *bz2* 模組中), 584
compress() (於 *compression.zstd* 模組中), 565
compress() (於 *gzip* 模組中), 579
compress() (於 *itertools* 模組中), 413
compress() (於 *lzma* 模組中), 589
compress() (於 *zlib* 模組中), 574
compress() (*zlib.Compress* 的方法), 576
compress_size (*zipfile.ZipInfo* 的屬性), 601
compress_type (*zipfile.ZipInfo* 的屬性), 600
compressed (*ipaddress.IPv4Address* 的屬性), 1544
compressed (*ipaddress.IPv4Network* 的屬性), 1550
compressed (*ipaddress.IPv6Address* 的屬性), 1547
compressed (*ipaddress.IPv6Network* 的屬性), 1553
compression() (*ssl.SSLSocket* 的方法), 1217
compression_level (*compression.zstd.CompressionParameter* 的屬性), 569
COMPRESSION_LEVEL_DEFAULT (於 *compression.zstd* 模組中), 573
CompressionError, 604
CompressionParameter (*compression.zstd* 中的類🄵), 569
compression.zstd
    module, 563
compressobj() (於 *zlib* 模組中), 575
COMSPEC, 725, 1049
concat() (於 *operator* 模組中), 437
Concatenate (於 *typing* 模組中), 1682
concatenation (串接)
    operation (操作), 46
concurrent.futures
    module, 1031
concurrent.interpreters
    module, 1039
cond (*bdb.Breakpoint* 的屬性), 1860
Condition (*asyncio* 中的類🄵), 1107
Condition (*multiprocessing* 中的類🄵), 999
condition (*pdb command*), 1873
Condition (*threading* 中的類🄵), 977
Condition() (*multiprocessing.managers.SyncManager* 的方法), 1006
config() (*tkinter.font.Font* 的方法), 1592
configparser
    module, 626
ConfigParser (*configparser* 中的類🄵), 638
configuration information (設定資訊), 1951
configuration (設定)
    file (檔案), 626
    file (檔案), debugger (偵錯器), 1872
    file (檔案), path (路徑), 2044
configure() (*tkinter.ttk.Style* 的方法), 1612
configure_mock() (*unittest.mock.Mock* 的方法), 1782
CONFORM (*enum.FlagBoundary* 的屬性), 331
confstr() (於 *os* 模組中), 731
confstr_names (於 *os* 模組中), 731
conjugate() (*decimal.Decimal* 的方法), 359
conjugate() (*numbers.Complex* 的方法), 337
conjugate() (🄵數方法), 39
connect() (*ftplib.FTP* 的方法), 1475
connect() (*http.client.HTTPConnection* 的方法), 1470
connect() (*multiprocessing.managers.BaseManager* 的方法), 1005
connect() (*smtplib.SMTP* 的方法), 1493
connect() (*socket.socket* 的方法), 1192
connect() (於 *sqlite3* 模組中), 534
connect_accepted_socket() (*asyncio.loop* 的方法), 1130
connect_ex() (*socket.socket* 的方法), 1192
connect_read_pipe() (*asyncio.loop* 的方法), 1134
connect_write_pipe() (*asyncio.loop* 的方法), 1134
Connection (*multiprocessing.connection* 中的類🄵), 997
Connection (*sqlite3* 中的類🄵), 538
connection (*sqlite3.Cursor* 的屬性), 550
connection_lost() (*asyncio.BaseProtocol* 的方法), 1152
connection_made() (*asyncio.BaseProtocol* 的方法), 1152
ConnectionAbortedError, 116
ConnectionError, 116
ConnectionRefusedError, 116
ConnectionResetError, 116
ConnectRegistry() (於 *winreg* 模組中), 2180
const (*optparse.Option* 的屬性), 912
Constant (*ast* 中的類🄵), 2098
constructor() (於 *copyreg* 模組中), 520
consumed (*asyncio.LimitOverrunError* 的屬性), 1119
Container (*collections.abc* 中的類🄵), 283
Container (*typing* 中的類🄵), 1717
container (容器)
    iteration over (🄵代於), 45
contains() (於 *operator* 模組中), 437
CONTAINS_OP (*opcode*), 2165
content (*urllib.error.ContentTooShortError* 的屬性), 1462
content type (🄵容類型)
    MIME, 1348
content_disposition (*email.headerregistry.ContentDispositionHeader* 的屬性), 1289

content_manager (*email.policy.EmailPolicy* 的屬性), 1282
content_size_flag (*compression.zstd.CompressionParameter* 的屬性), 571
content_type (*email.headerregistry.ContentTypeHeader* 的屬性), 1289
ContentDispositionHeader (*email.headerregistry* 中的類F), 1289
ContentHandler (*xml.sax.handler* 中的類F), 1402
ContentManager (*email.contentmanager* 中的類F), 1292
contents (*ctypes._Pointer* 的屬性), 853
contents() (*importlib.abc.ResourceReader* 的方法), 2067
contents() (*importlib.resources.abc.ResourceReader* 的方法), 2082
contents() (於 *importlib.resources* 模組中), 2081
ContentTooShortError, 1462
ContentTransferEncoding (*email.headerregistry* 中的類F), 1289
ContentTypeHeader (*email.headerregistry* 中的類F), 1289
Context (*contextvars* 中的類F), 1069
Context (*decimal* 中的類F), 366
context (*ssl.SSLSocket* 的屬性), 1218
context management protocol（情境管理協定）, 96, **2230**
context manager（情境管理器）, 96, **2230**
context variable（情境變數）, **2230**
context_aware_warnings (*sys.flags* 的屬性), 1926
context_diff() (於 *difflib* 模組中), 162
ContextDecorator (*contextlib* 中的類F), 1985
contextlib
    module, 1980
ContextManager (*typing* 中的類F), 1719
contextmanager() (於 *contextlib* 模組中), 1980
ContextVar (*contextvars* 中的類F), 1068
contextvars
    module, 1067
context（情境）, **2230**
CONTIG (*inspect.BufferFlags* 的屬性), 2034
CONTIG_RO (*inspect.BufferFlags* 的屬性), 2034
contiguous (*memoryview* 的屬性), 89
contiguous（連續的）, **2230**
Continue (*ast* 中的類F), 2111
CONTINUE (*compression.zstd.ZstdCompressor* 的屬性), 566
continue (*pdb command*), 1874
CONTINUOUS (*enum.EnumCheck* 的屬性), 330
control() (*select.kqueue* 的方法), 1242
controlnames (於 *curses.ascii* 模組中), 958
CONTTYPE (於 *tarfile* 模組中), 605
conversion (*string.templatelib.Interpolation* 的屬性), 138
conversions（轉F）
    numeric（數值）, 39
convert() (於 *string.templatelib* 模組中), 139
convert_arg_line_to_args() (*argparse.ArgumentParser* 的方法), 883
convert_field() (*string.Formatter* 的方法), 125
CONVERT_VALUE (*opcode*), 2170
Cookie (*http.cookiejar* 中的類F), 1523
CookieError, 1519
CookieJar (*http.cookiejar* 中的類F), 1522
cookiejar (*urllib.request.HTTPCookieProcessor* 的屬性), 1445
CookiePolicy (*http.cookiejar* 中的類F), 1522
Coordinated Universal Time（世界協調時間）, 748
--copies
    venv 命令列選項, 1906
copy
    module, 309
COPY (*opcode*), 2157
copy() (*collections.deque* 的方法), 269
copy() (*contextvars.Context* 的方法), 1070
copy() (*decimal.Context* 的方法), 367
copy() (*dict* 的方法), 93
copy() (*frozenset* 的方法), 90
copy() (*hashlib.hash* 的方法), 653
copy() (*hmac.HMAC* 的方法), 663
copy() (*http.cookies.Morsel* 的方法), 1521
copy() (*imaplib.IMAP4* 的方法), 1485
copy() (*pathlib.Path* 的方法), 464
copy() (*sequence* 的方法), 49
copy() (*tkinter.font.Font* 的方法), 1592
copy() (*types.MappingProxyType* 的方法), 308
copy() (於 *copy* 模組中), 309
copy() (於 *multiprocessing.sharedctypes* 模組中), 1003
copy() (於 *shutil* 模組中), 494
copy() (*zlib.Compress* 的方法), 576
copy() (*zlib.Decompress* 的方法), 577
copy2() (於 *shutil* 模組中), 494
copy_abs() (*decimal.Context* 的方法), 368
copy_abs() (*decimal.Decimal* 的方法), 359
copy_context() (於 *contextvars* 模組中), 1069
copy_decimal() (*decimal.Context* 的方法), 367
copy_file_range() (於 *os* 模組中), 677
COPY_FREE_VARS (*opcode*), 2168
copy_into() (*pathlib.Path* 的方法), 464
copy_location() (於 *ast* 模組中), 2126
copy_negate() (*decimal.Context* 的方法), 368
copy_negate() (*decimal.Decimal* 的方法), 359
copy_sign() (*decimal.Context* 的方法), 368
copy_sign() (*decimal.Decimal* 的方法), 359
CopyComPointer() (於 *ctypes* 模組中), 842
copyfile() (於 *shutil* 模組中), 493
copyfileobj() (於 *shutil* 模組中), 493
copying files（FF檔案）, 492
copymode() (於 *shutil* 模組中), 493
copyreg
    module, 520
copyright (F建變數), 36
copyright (於 *sys* 模組中), 1921
copysign() (於 *math* 模組中), 343

copystat() (於 shutil 模組中), 493
copytree() (於 shutil 模組中), 495
Copy（🄵🄵）, 1621
copy（🄵🄵）
    module（模組）, 520
    protocol（協定）, 512
Coroutine (collections.abc 中的類🄵), 284
Coroutine (typing 中的類🄵), 1718
coroutine function（協程函式）, **2230**
coroutine() (於 types 模組中), 309
CoroutineType (於 types 模組中), 305
coroutine（協程）, **2230**
correlation() (於 statistics 模組中), 402
cos() (於 cmath 模組中), 351
cos() (於 math 模組中), 347
cosh() (於 cmath 模組中), 351
cosh() (於 math 模組中), 347
--count
    trace 命令列選項, 1891
    uuid 命令列選項, 1501
count (tracemalloc.Statistic 的屬性), 1901
count (tracemalloc.StatisticDiff 的屬性), 1901
count() (array.array 的方法), 295
count() (bytearray 的方法), 70
count() (bytes 的方法), 70
count() (collections.deque 的方法), 269
count() (memoryview 的方法), 87
count() (multiprocessing.shared_memory.ShareableList 的方法), 1030
count() (sequence 的方法), 47
count() (str 的方法), 54
count() (於 itertools 模組中), 413
count_diff (tracemalloc.StatisticDiff 的屬性), 1901
Counter (collections 中的類🄵), 266
Counter (typing 中的類🄵), 1715
countOf() (於 operator 模組中), 438
countTestCases() (unittest.TestCase 的方法), 1764
countTestCases() (unittest.TestSuite 的方法), 1767
covariance() (於 statistics 模組中), 402
CoverageResults (trace 中的類🄵), 1892
--coverdir
    trace 命令列選項, 1891
cProfile
    module, 1880
cProfile 命令列選項
    -m, 1879
    -o, 1879
    -s, 1879
CPU time（CPU 時間）, 751, 756
cpu_count() (於 multiprocessing 模組中), 995
cpu_count() (於 os 模組中), 731
CPython, **2231**
cpython_only() (於 test.support 模組中), 1842
CR (於 curses.ascii 模組中), 955
crawl_delay() (urllib.robotparser.RobotFileParser 的方法), 1462
CRC (zipfile.ZipInfo 的屬性), 601
crc32() (於 binascii 模組中), 1357
crc32() (於 zlib 模組中), 575
crc_hqx() (於 binascii 模組中), 1357
--create
    tarfile 命令列選項, 616
    zipfile 命令列選項, 601
create() (imaplib.IMAP4 的方法), 1485
create() (venv.EnvBuilder 的方法), 1908
create() (於 concurrent.interpreters 模組中), 1042
create() (於 venv 模組中), 1910
create_aggregate() (sqlite3.Connection 的方法), 539
create_archive() (於 zipapp 模組中), 1915
create_autospec() (於 unittest.mock 模組中), 1809
CREATE_BREAKAWAY_FROM_JOB (於 subprocess 模組中), 1057
create_collation() (sqlite3.Connection 的方法), 541
create_configuration() (venv.EnvBuilder 的方法), 1909
create_connection() (asyncio.loop 的方法), 1126
create_connection() (於 socket 模組中), 1185
create_datagram_endpoint() (asyncio.loop 的方法), 1127
create_decimal() (decimal.Context 的方法), 367
create_decimal_from_float() (decimal.Context 的方法), 367
create_default_context() (於 ssl 模組中), 1204
CREATE_DEFAULT_ERROR_MODE (於 subprocess 模組中), 1057
create_eager_task_factory() (於 asyncio 模組中), 1087
create_empty_file() (於 test.support.os_helper 模組中), 1850
create_function() (sqlite3.Connection 的方法), 539
create_future() (asyncio.loop 的方法), 1125
create_git_ignore_file() (venv.EnvBuilder 的方法), 1910
create_module() (importlib.abc.Loader 的方法), 2063
create_module() (importlib.machinery.ExtensionFileLoader 的方法), 2071
create_module() (zipimport.zipimporter 的方法), 2052
CREATE_NEW_CONSOLE (於 subprocess 模組中), 1056
CREATE_NEW_PROCESS_GROUP (於 subprocess 模組中), 1056
CREATE_NO_WINDOW (於 subprocess 模組中), 1057
create_queue() (於 concurrent.interpreters 模組中), 1042
create_server() (asyncio.loop 的方法), 1128
create_server() (於 socket 模組中), 1185
create_stats() (profile.Profile 的方法), 1881
create_string_buffer() (於 ctypes 模組中), 842
create_subprocess_exec() (於 asyncio 模組中), 1111
create_subprocess_shell() (於 asyncio 模組中),

1112
create_system (*zipfile.ZipInfo* 的屬性), 600
create_task() (*asyncio.loop* 的方法), 1125
create_task() (*asyncio.TaskGroup* 的方法), 1083
create_task() (於 *asyncio* 模組中), 1082
create_unicode_buffer() (於 *ctypes* 模組中), 843
create_unix_connection() (*asyncio.loop* 的方法), 1128
create_unix_server() (*asyncio.loop* 的方法), 1130
create_version (*zipfile.ZipInfo* 的屬性), 600
create_window_function() (*sqlite3.Connection* 的方法), 540
createAttribute() (*xml.dom.Document* 的方法), 1389
createAttributeNS() (*xml.dom.Document* 的方法), 1389
createComment() (*xml.dom.Document* 的方法), 1389
createDocument() (*xml.dom.DOMImplementation* 的方法), 1386
createDocumentType() (*xml.dom.DOMImplementation* 的方法), 1386
createElement() (*xml.dom.Document* 的方法), 1389
createElementNS() (*xml.dom.Document* 的方法), 1389
createfilehandler() (*_tkinter.Widget.tk* 的方法), 1590
CreateKey() (於 *winreg* 模組中), 2180
CreateKeyEx() (於 *winreg* 模組中), 2180
createLock() (*logging.Handler* 的方法), 766
createLock() (*logging.NullHandler* 的方法), 792
createProcessingInstruction() (*xml.dom.Document* 的方法), 1389
createSocket() (*logging.handlers.SocketHandler* 的方法), 796
createSocket() (*logging.handlers.SysLogHandler* 的方法), 798
createTextNode() (*xml.dom.Document* 的方法), 1389
credits (🄵建變數), 36
CRITICAL (於 *logging* 模組中), 766
critical() (*logging.Logger* 的方法), 764
critical() (於 *logging* 模組中), 774
CRNCYSTR (於 *locale* 模組中), 1572
CRT_ASSEMBLY_VERSION (於 *msvcrt* 模組中), 2179
CRT_ASSERT (於 *msvcrt* 模組中), 2179
CRT_ERROR (於 *msvcrt* 模組中), 2179
CRT_WARN (於 *msvcrt* 模組中), 2179
CRTDBG_MODE_DEBUG (於 *msvcrt* 模組中), 2179
CRTDBG_MODE_FILE (於 *msvcrt* 模組中), 2179
CRTDBG_MODE_WNDW (於 *msvcrt* 模組中), 2179
CRTDBG_REPORT_MODE (於 *msvcrt* 模組中), 2179
CrtSetReportFile() (於 *msvcrt* 模組中), 2179
CrtSetReportMode() (於 *msvcrt* 模組中), 2179
crypt
    module, 2220
cryptography（密碼學）, 651
--css
    calendar 命令列選項, 263
cssclass_month (*calendar.HTMLCalendar* 的屬性), 258
cssclass_month_head (*calendar.HTMLCalendar* 的屬性), 258
cssclass_noday (*calendar.HTMLCalendar* 的屬性), 258
cssclass_year (*calendar.HTMLCalendar* 的屬性), 258
cssclass_year_head (*calendar.HTMLCalendar* 的屬性), 259
cssclasses (*calendar.HTMLCalendar* 的屬性), 258
cssclasses_weekday_head (*calendar.HTMLCalendar* 的屬性), 258
csv, 619
    module, 619
cte (*email.headerregistry.ContentTransferEncoding* 的屬性), 1289
cte_type (*email.policy.Policy* 的屬性), 1279
ctermid() (於 *os* 模組中), 669
ctime() (*datetime.date* 的方法), 223
ctime() (*datetime.datetime* 的方法), 233
ctime() (於 *time* 模組中), 749
ctrl() (於 *curses.ascii* 模組中), 957
CTRL_BREAK_EVENT (於 *signal* 模組中), 1250
CTRL_C_EVENT (於 *signal* 模組中), 1250
ctypes
    module, 817
curdir (於 *os* 模組中), 732
currency() (於 *locale* 模組中), 1574
current context, **2231**
current() (*tkinter.ttk.Combobox* 的方法), 1602
current_process() (於 *multiprocessing* 模組中), 996
current_task() (於 *asyncio* 模組中), 1094
current_thread() (於 *threading* 模組中), 967
CurrentByteIndex (*xml.parsers.expat.xmlparser* 的屬性), 1414
CurrentColumnNumber (*xml.parsers.expat.xmlparser* 的屬性), 1414
currentframe() (於 *inspect* 模組中), 2031
CurrentLineNumber (*xml.parsers.expat.xmlparser* 的屬性), 1414
curs_set() (於 *curses* 模組中), 930
curses
    module, 928
curses.ascii
    module, 954
curses.panel
    module, 958
curses.textpad
    module, 953
Cursor (*sqlite3* 中的類🄵), 548
cursor() (*sqlite3.Connection* 的方法), 538
cursyncup() (*curses.window* 的方法), 937
Cut（剪下）, 1621
cwd() (*ftplib.FTP* 的方法), 1477
cwd() (*pathlib.Path* 的類🄵方法), 456

cycle() (於 itertools 模組中), 414
CycleError, 336
cyclic isolate, 2231
Cyclic Redundancy Check（循環冗余核對）, 575

D

-d
    compileall 命令列選項, 2147
    gzip 命令列選項, 581
    http.server 命令列選項, 1517
    idle 命令列選項, 1624
D_FMT (於 locale 模組中), 1571
D_T_FMT (於 locale 模組中), 1571
daemon (multiprocessing.Process 的屬性), 990
daemon (threading.Thread 的屬性), 973
daemon_threads (socketserver.ThreadingMixIn 的屬性), 1504
data (collections.UserDict 的屬性), 279
data (collections.UserList 的屬性), 279
data (collections.UserString 的屬性), 279
data (plistlib.UID 的屬性), 648
data (select.kevent 的屬性), 1243
data (selectors.SelectorKey 的屬性), 1245
data (urllib.request.Request 的屬性), 1441
data (xml.dom.Comment 的屬性), 1391
data (xml.dom.ProcessingInstruction 的屬性), 1392
data (xml.dom.Text 的屬性), 1391
data (xmlrpc.client.Binary 的屬性), 1534
data() (xml.etree.ElementTree.TreeBuilder 的方法), 1381
data_filter() (於 tarfile 模組中), 613
data_open() (urllib.request.DataHandler 的方法), 1447
data_received() (asyncio.Protocol 的方法), 1152
DatabaseError, 552
databases（資料庫）, 530
database（資料庫）
    Unicode, 176
dataclass() (於 dataclasses 模組中), 1970
dataclass_transform() (於 typing 模組中), 1707
dataclasses
    module, 1969
DataError, 552
datagram_received() (asyncio.DatagramProtocol 的方法), 1154
DatagramHandler (logging.handlers 中的類F), 796
DatagramProtocol (asyncio 中的類F), 1152
DatagramRequestHandler (socketserver 中的類F), 1507
DatagramTransport (asyncio 中的類F), 1148
DataHandler (urllib.request 中的類F), 1440
data（資料）
    packing（打包） binary（二進位）, 187
    tabular（表格）, 619
date (datetime 中的類F), 219
date() (datetime.datetime 的方法), 230
date_time (zipfile.ZipInfo 的屬性), 599
date_time_string() (http.server.BaseHTTPRequestHandler 的方法), 1515
DateHeader (email.headerregistry 中的類F), 1287
datetime
    module, 213
datetime (datetime 中的類F), 224
datetime (email.headerregistry.DateHeader 的屬性), 1288
DateTime (xmlrpc.client 中的類F), 1533
Day (calendar 中的類F), 260
day (datetime.date 的屬性), 221
day (datetime.datetime 的屬性), 228
DAY_1 (於 locale 模組中), 1571
DAY_2 (於 locale 模組中), 1571
DAY_3 (於 locale 模組中), 1571
DAY_4 (於 locale 模組中), 1571
DAY_5 (於 locale 模組中), 1571
DAY_6 (於 locale 模組中), 1571
DAY_7 (於 locale 模組中), 1571
day_abbr (於 calendar 模組中), 260
day_name (於 calendar 模組中), 260
daylight (於 time 模組中), 759
Daylight Saving Time（日光節約時間）, 748
days (datetime.timedelta 的屬性), 217
DbfilenameShelf (shelve 中的類F), 523
dbm
    module, 525
dbm.dumb
    module, 530
dbm.gnu
    module, 528
    module（模組）, 522
dbm.ndbm
    module, 529
    module（模組）, 522
dbm.sqlite3
    module, 527
DC1 (於 curses.ascii 模組中), 956
DC2 (於 curses.ascii 模組中), 956
DC3 (於 curses.ascii 模組中), 956
DC4 (於 curses.ascii 模組中), 956
dcgettext() (於 locale 模組中), 1577
deactivate_stack_trampoline() (於 sys 模組中), 1942
debug (imaplib.IMAP4 的屬性), 1490
debug (pdb command), 1877
debug (shlex.shlex 的屬性), 2194
debug (sys.flags 的屬性), 1926
DEBUG (於 logging 模組中), 766
DEBUG (於 re 模組中), 146
debug (zipfile.ZipFile 的屬性), 597
debug() (logging.Logger 的方法), 763
debug() (unittest.TestCase 的方法), 1757
debug() (unittest.TestSuite 的方法), 1767
debug() (於 doctest 模組中), 1744
debug() (於 logging 模組中), 774

DEBUG_BYTECODE_SUFFIXES (於 *importlib.machinery* 模組中), 2068
DEBUG_COLLECTABLE (於 *gc* 模組中), 2015
DEBUG_LEAK (於 *gc* 模組中), 2016
DEBUG_SAVEALL (於 *gc* 模組中), 2016
debug_src() (於 *doctest* 模組中), 1745
DEBUG_STATS (於 *gc* 模組中), 2015
DEBUG_UNCOLLECTABLE (於 *gc* 模組中), 2015
debugger (偵錯器)
    configuration (設定) file (檔案), 1872
debugger (除錯器), 968, 1620, 1931, 1940
debugging (偵錯), 1868
debuglevel (*http.client.HTTPResponse* 的屬性), 1472
DebugRunner (*doctest* 中的類🄵), 1745
DECEMBER (於 *calendar* 模組中), 261
decimal
    module, 353
Decimal (*decimal* 中的類🄵), 357
decimal() (於 *unicodedata* 模組中), 177
DecimalException (*decimal* 中的類🄵), 372
decode (*codecs.CodecInfo* 的屬性), 194
decode() (*bytearray* 的方法), 71
decode() (*bytes* 的方法), 71
decode() (*codecs.Codec* 的方法), 199
decode() (*codecs.IncrementalDecoder* 的方法), 200
decode() (*json.JSONDecoder* 的方法), 1324
decode() (於 *base64* 模組中), 1355
decode() (於 *codecs* 模組中), 194
decode() (於 *quopri* 模組中), 1358
decode() (*xmlrpc.client.Binary* 的方法), 1534
decode() (*xmlrpc.client.DateTime* 的方法), 1533
decode_header() (於 *email.header* 模組中), 1312
decode_params() (於 *email.utils* 模組中), 1318
decode_rfc2231() (於 *email.utils* 模組中), 1318
decode_source() (於 *importlib.util* 模組中), 2074
decodebytes() (於 *base64* 模組中), 1355
DecodedGenerator (*email.generator* 中的類🄵), 1277
decodestring() (於 *quopri* 模組中), 1358
decode (解碼)
    Codecs, 194
decomposition() (於 *unicodedata* 模組中), 178
--decompress
    gzip 命令列選項, 581
decompress() (*bz2.BZ2Decompressor* 的方法), 583
decompress() (*compression.zstd.ZstdDecompressor* 的方法), 567
decompress() (*lzma.LZMADecompressor* 的方法), 588
decompress() (於 *bz2* 模組中), 584
decompress() (於 *compression.zstd* 模組中), 565
decompress() (於 *gzip* 模組中), 580
decompress() (於 *lzma* 模組中), 589
decompress() (於 *zlib* 模組中), 575
decompress() (*zlib.Decompress* 的方法), 576
decompressed_size (*compression.zstd.FrameInfo* 的屬性), 573
DecompressionParameter (*compression.zstd* 中的類🄵), 572
decompressobj() (於 *zlib* 模組中), 576
decorator (裝飾器), **2231**
DEDENT (於 *token* 模組中), 2134
dedent() (於 *textwrap* 模組中), 173
deepcopy() (於 *copy* 模組中), 310
def_prog_mode() (於 *curses* 模組中), 930
def_shell_mode() (於 *curses* 模組中), 930
default (*inspect.Parameter* 的屬性), 2024
default (*optparse.Option* 的屬性), 912
default (於 *email.policy* 模組中), 1283
DEFAULT (於 *unittest.mock* 模組中), 1807
default() (*cmd.Cmd* 的方法), 960
default() (*json.JSONEncoder* 的方法), 1326
DEFAULT_BUFFER_SIZE (於 *io* 模組中), 735
default_bufsize (於 *xml.dom.pulldom* 模組中), 1399
default_exception_handler() (*asyncio.loop* 的方法), 1136
default_factory (*collections.defaultdict* 的屬性), 272
DEFAULT_FORMAT (於 *tarfile* 模組中), 606
DEFAULT_IGNORES (於 *filecmp* 模組中), 482
default_loader() (於 *xml.etree.ElementInclude* 模組中), 1376
default_max_str_digits (*sys.int_info* 的屬性), 1934
default_open() (*urllib.request.BaseHandler* 的方法), 1443
DEFAULT_PROTOCOL (於 *pickle* 模組中), 507
DEFAULT_TIMEOUT (*unittest.mock.ThreadingMock* 的屬性), 1792
default_timer() (於 *timeit* 模組中), 1887
DefaultContext (於 *decimal* 模組中), 366
DefaultCookiePolicy (*http.cookiejar* 中的類🄵), 1523
defaultdict (*collections* 中的類🄵), 272
DefaultDict (*typing* 中的類🄵), 1715
DefaultEventLoopPolicy (*asyncio* 中的類🄵), 1161
DefaultHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
DefaultHandlerExpand() (*xml.parsers.expat.xmlparser* 的方法), 1416
--default-pip
    ensurepip 命令列選項, 1904
defaults() (*configparser.ConfigParser* 的方法), 639
DefaultSelector (*selectors* 中的類🄵), 1246
defaultTestLoader (於 *unittest* 模組中), 1772
defaultTestResult() (*unittest.TestCase* 的方法), 1764
defects (*email.headerregistry.BaseHeader* 的屬性), 1287
defects (*email.message.EmailMessage* 的屬性), 1271
defects (*email.message.Message* 的屬性), 1308
defpath (於 *os* 模組中), 732
DefragResult (*urllib.parse* 中的類🄵), 1458
DefragResultBytes (*urllib.parse* 中的類🄵), 1459

degrees() (於 *math* 模組中), 346
degrees() (於 *turtle* 模組中), 1641
del
    statement（陳述式）, 48, 91
Del (*ast* 中的類🄵), 2101
DEL (於 *curses.ascii* 模組中), 956
del_param() (*email.message.EmailMessage* 的方法), 1268
del_param() (*email.message.Message* 的方法), 1306
delattr()
    built-in function, 12
delay() (於 *turtle* 模組中), 1654
delay_output() (於 *curses* 模組中), 930
delayload (*http.cookiejar.FileCookieJar* 的屬性), 1525
delch() (*curses.window* 的方法), 937
dele() (*poplib.POP3* 的方法), 1481
Delete (*ast* 中的類🄵), 2108
delete() (*ftplib.FTP* 的方法), 1477
delete() (*imaplib.IMAP4* 的方法), 1485
delete() (*tkinter.ttk.Treeview* 的方法), 1610
DELETE_ATTR (*opcode*), 2163
DELETE_DEREF (*opcode*), 2168
DELETE_FAST (*opcode*), 2167
DELETE_GLOBAL (*opcode*), 2163
DELETE_NAME (*opcode*), 2162
DELETE_SUBSCR (*opcode*), 2159
deleteacl() (*imaplib.IMAP4* 的方法), 1485
deletefilehandler() (*_tkinter.Widget.tk* 的方法), 1591
DeleteKey() (於 *winreg* 模組中), 2181
DeleteKeyEx() (於 *winreg* 模組中), 2181
deleteln() (*curses.window* 的方法), 937
deleteMe() (*bdb.Breakpoint* 的方法), 1859
DeleteValue() (於 *winreg* 模組中), 2181
delimiter (*csv.Dialect* 的屬性), 623
delitem() (於 *operator* 模組中), 438
deliver_challenge() (於 *multiprocessing.connection* 模組中), 1013
delocalize() (於 *locale* 模組中), 1574
demo_app() (於 *wsgiref.simple_server* 模組中), 1429
denominator (*fractions.Fraction* 的屬性), 382
denominator (*numbers.Rational* 的屬性), 338
deprecated() (於 *warnings* 模組中), 1968
DeprecationWarning, 117
deque (*collections* 中的類🄵), 269
Deque (*typing* 中的類🄵), 1715
dequeue() (*logging.handlers.QueueListener* 的方法), 803
DER_cert_to_PEM_cert() (於 *ssl* 模組中), 1207
derive() (*BaseExceptionGroup* 的方法), 119
derwin() (*curses.window* 的方法), 938
description (*inspect.Parameter.kind* 的屬性), 2025
description (*sqlite3.Cursor* 的屬性), 550
descriptor（描述器）, **2231**
deserialize() (*sqlite3.Connection* 的方法), 546
dest (*optparse.Option* 的屬性), 912
detach() (*io.BufferedIOBase* 的方法), 739
detach() (*io.TextIOBase* 的方法), 743
detach() (*socket.socket* 的方法), 1192
detach() (*tkinter.ttk.Treeview* 的方法), 1610
detach() (*weakref.finalize* 的方法), 299
Detach() (*winreg.PyHKEY* 的方法), 2188
DETACHED_PROCESS (於 *subprocess* 模組中), 1057
--details
    inspect 命令列選項, 2035
details (*ctypes.COMError* 的屬性), 853
detect_api_mismatch() (於 *test.support* 模組中), 1844
detect_encoding() (於 *tokenize* 模組中), 2140
deterministic profiling, 1878
dev_mode (*sys.flags* 的屬性), 1926
device_encoding() (於 *os* 模組中), 678
devmajor (*tarfile.TarInfo* 的屬性), 611
devminor (*tarfile.TarInfo* 的屬性), 611
devnull (於 *os* 模組中), 732
DEVNULL (於 *subprocess* 模組中), 1046
devpoll() (於 *select* 模組中), 1237
DevpollSelector (*selectors* 中的類🄵), 1246
dfast (*compression.zstd.Strategy* 的屬性), 572
dgettext() (於 *gettext* 模組中), 1562
dgettext() (於 *locale* 模組中), 1577
Dialect (*csv* 中的類🄵), 621
dialect (*csv.csvreader* 的屬性), 624
dialect (*csv.csvwriter* 的屬性), 625
Dialog (*tkinter.commondialog* 中的類🄵), 1595
Dialog (*tkinter.simpledialog* 中的類🄵), 1593
Dict (*ast* 中的類🄵), 2100
Dict (*typing* 中的類🄵), 1714
dict (🄵建類🄵), 91
dict() (*multiprocessing.managers.SyncManager* 的方法), 1006
dict_content (*compression.zstd.ZstdDict* 的屬性), 568
dict_id (*compression.zstd.ZstdDict* 的屬性), 568
dict_id_flag (*compression.zstd.CompressionParameter* 的屬性), 571
DICT_MERGE (*opcode*), 2165
DICT_UPDATE (*opcode*), 2164
DictComp (*ast* 中的類🄵), 2105
dictConfig() (於 *logging.config* 模組中), 779
dictionary comprehension（字典綜合運算）, **2231**
dictionary view（字典檢視）, **2231**
dictionary_id (*compression.zstd.FrameInfo* 的屬性), 573
dictionary（字典）, **2231**
    object（物件）, 91
    type（型🄵）, operations on（操作於）, 91
DictReader (*csv* 中的類🄵), 620
DictWriter (*csv* 中的類🄵), 621
diff_bytes() (於 *difflib* 模組中), 164
diff_files (*filecmp.dircmp* 的屬性), 482
Differ (*difflib* 中的類🄵), 161
difference() (*frozenset* 的方法), 90

difference_update() (*frozenset* 的方法), 91
difflib
    module, 161
dig (*sys.float_info* 的屬性), 1928
digest() (*hashlib.hash* 的方法), 653
digest() (*hashlib.shake* 的方法), 654
digest() (*hmac.HMAC* 的方法), 662
digest() (於 *hmac* 模組中), 662
digest_size (*hmac.HMAC* 的屬性), 663
digit() (於 *unicodedata* 模組中), 177
digits (於 *string* 模組中), 123
dir()
    built-in function, 13
dir() (*ftplib.FTP* 的方法), 1477
dircmp (*filecmp* 中的類🄵), 481
directory
    compileall 命令列選項, 2147
--directory
    http.server 命令列選項, 1517
Directory (*tkinter.filedialog* 中的類🄵), 1594
directory (目🄵)
    changing (改變), 691
    creating (建立), 696
    deleting (🄵除), 495, 698
    site-packages, 2043
    traversal (遍歷), 708, 709
    walking, 708, 709
DirEntry (*os* 中的類🄵), 700
dirname() (於 *os.path* 模組中), 470
dirs_double_event() (*tkinter.filedialog.FileDialog* 的方法), 1594
dirs_select_event() (*tkinter.filedialog.FileDialog* 的方法), 1594
DirsOnSysPath (*test.support.import_helper* 中的類🄵), 1852
DIRTYPE (於 *tarfile* 模組中), 605
dis
    module, 2151
dis 命令列選項
    -C, 2152
    -h, 2152
    --help, 2152
    -O, 2152
    -P, 2152
    -S, 2152
    --show-caches, 2152
    --show-offsets, 2152
    --show-positions, 2152
    --specialized, 2152
dis() (*dis.Bytecode* 的方法), 2153
dis() (於 *dis* 模組中), 2154
dis() (於 *pickletools* 模組中), 2175
disable (*pdb command*), 1873
DISABLE (於 *sys.monitoring* 模組中), 1950
disable() (*bdb.Breakpoint* 的方法), 1859
disable() (*profile.Profile* 的方法), 1881
disable() (於 *faulthandler* 模組中), 1866
disable() (於 *gc* 模組中), 2012
disable() (於 *logging* 模組中), 774
disable_current_event() (*bdb.Bdb* 的方法), 1863
disable_faulthandler() (於 *test.support* 模組中), 1840
disable_gc() (於 *test.support* 模組中), 1841
disable_interspersed_args() (*optparse.OptionParser* 的方法), 916
disabled (*logging.Logger* 的屬性), 762
DisableReflectionKey() (於 *winreg* 模組中), 2185
disassemble() (於 *dis* 模組中), 2154
discard (*http.cookiejar.Cookie* 的屬性), 1529
discard() (*frozenset* 的方法), 91
discard() (*mailbox.Mailbox* 的方法), 1330
discard() (*mailbox.MH* 的方法), 1337
disco() (於 *dis* 模組中), 2154
discover() (*unittest.TestLoader* 的方法), 1769
disk_usage() (於 *shutil* 模組中), 497
dispatch_call() (*bdb.Bdb* 的方法), 1861
dispatch_exception() (*bdb.Bdb* 的方法), 1862
dispatch_line() (*bdb.Bdb* 的方法), 1861
dispatch_return() (*bdb.Bdb* 的方法), 1861
dispatch_table (*pickle.Pickler* 的屬性), 509
DISPLAY, 1581
display (*pdb command*), 1875
display_name (*email.headerregistry.Address* 的屬性), 1291
display_name (*email.headerregistry.Group* 的屬性), 1291
displayhook() (於 *sys* 模組中), 1922
dist() (於 *math* 模組中), 346
distance() (於 *turtle* 模組中), 1641
distb() (於 *dis* 模組中), 2154
Distribution (*importlib.metadata* 中的類🄵), 2088
distribution() (於 *importlib.metadata* 模組中), 2088
distutils
    module, 2220
Div (*ast* 中的類🄵), 2102
divide() (*decimal.Context* 的方法), 368
divide_int() (*decimal.Context* 的方法), 368
division (於 *__future__* 模組中), 2011
DivisionByZero (*decimal* 中的類🄵), 372
divmod()
    built-in function, 13
divmod() (*decimal.Context* 的方法), 368
DLE (於 *curses.ascii* 模組中), 956
DllCanUnloadNow() (於 *ctypes* 模組中), 843
DllGetClassObject() (於 *ctypes* 模組中), 843
dllhandle (於 *sys* 模組中), 1922
dllist() (於 *ctypes.util* 模組中), 843
dnd_start() (於 *tkinter.dnd* 模組中), 1598
DndHandler (*tkinter.dnd* 中的類🄵), 1598
dngettext() (於 *gettext* 模組中), 1562
dnpgettext() (於 *gettext* 模組中), 1562
do_clear() (*bdb.Bdb* 的方法), 1862
do_command() (*curses.textpad.Textbox* 的方法), 954
do_GET() (*http.server.SimpleHTTPRequestHandler* 的方法), 1515

do_handshake() (*ssl.SSLSocket* 的方法), 1215
do_HEAD() (*http.server.SimpleHTTPRequestHandler* 的方法), 1515
do_help() (*cmd.Cmd* 的方法), 960
do_POST() (*http.server.CGIHTTPRequestHandler* 的方法), 1516
doc (*json.JSONDecodeError* 的屬性), 1326
doc (*tomllib.TOMLDecodeError* 的屬性), 645
doc_header (*cmd.Cmd* 的屬性), 961
DocCGIXMLRPCRequestHandler (*xmlrpc.server* 中的類🄵), 1542
DocFileSuite() (於 *doctest* 模組中), 1736
doClassCleanups() (*unittest.TestCase* 的類🄵方法), 1765
doCleanups() (*unittest.TestCase* 的方法), 1764
docmd() (*smtplib.SMTP* 的方法), 1492
docstring (*doctest.DocTest* 的屬性), 1739
docstring（🄵明字串）, **2231**
doctest
    module, 1724
DocTest (*doctest* 中的類🄵), 1738
doctest 命令列選項
    -f, 1727
    --fail-fast, 1727
    -o, 1727
    --option, 1727
    -v, 1727
    --verbose, 1727
DocTestFailure, 1745
DocTestFinder (*doctest* 中的類🄵), 1740
DocTestParser (*doctest* 中的類🄵), 1740
DocTestRunner (*doctest* 中的類🄵), 1741
DocTestSuite() (於 *doctest* 模組中), 1737
doctype() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
documentation（文件）
    generation（🄵生）, 1720
    online（🄵上）, 1720
documentElement (*xml.dom.Document* 的屬性), 1389
DocXMLRPCRequestHandler (*xmlrpc.server* 中的類🄵), 1542
DocXMLRPCServer (*xmlrpc.server* 中的類🄵), 1542
domain (*email.headerregistry.Address* 的屬性), 1291
domain (*http.cookiejar.Cookie* 的屬性), 1528
domain (*http.cookies.Morsel* 的屬性), 1520
domain (*tracemalloc.DomainFilter* 的屬性), 1899
domain (*tracemalloc.Filter* 的屬性), 1899
domain (*tracemalloc.Trace* 的屬性), 1901
domain_initial_dot (*http.cookiejar.Cookie* 的屬性), 1529
domain_return_ok() (*http.cookiejar.CookiePolicy* 的方法), 1526
domain_specified (*http.cookiejar.Cookie* 的屬性), 1529
DomainFilter (*tracemalloc* 中的類🄵), 1899
DomainLiberal (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
DomainRFC2965Match (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
DomainStrict (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
DomainStrictNoDots (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
DomainStrictNonDomain (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
DOMEventStream (*xml.dom.pulldom* 中的類🄵), 1399
DOMException, 1392
doModuleCleanups() (於 *unittest* 模組中), 1776
DomstringSizeErr, 1392
done() (*asyncio.Future* 的方法), 1145
done() (*asyncio.Task* 的方法), 1094
done() (*concurrent.futures.Future* 的方法), 1037
done() (*graphlib.TopologicalSorter* 的方法), 335
done() (於 *turtle* 模組中), 1656
DONT_ACCEPT_BLANKLINE (於 *doctest* 模組中), 1731
DONT_ACCEPT_TRUE_FOR_1 (於 *doctest* 模組中), 1731
dont_write_bytecode (*sys.flags* 的屬性), 1926
dont_write_bytecode (於 *sys* 模組中), 1922
doRollover() (*logging.handlers.RotatingFileHandler* 的方法), 794
doRollover() (*logging.handlers.TimedRotatingFileHandler* 的方法), 795
DOT (於 *token* 模組中), 2137
dot() (於 *turtle* 模組中), 1638
DOTALL (於 *re* 模組中), 147
doublequote (*csv.Dialect* 的屬性), 623
DOUBLESLASH (於 *token* 模組中), 2138
DOUBLESLASHEQUAL (於 *token* 模組中), 2138
DOUBLESTAR (於 *token* 模組中), 2137
DOUBLESTAREQUAL (於 *token* 模組中), 2138
doupdate() (於 *curses* 模組中), 930
down (*pdb command*), 1872
down() (於 *turtle* 模組中), 1642
dpgettext() (於 *gettext* 模組中), 1562
drain() (*asyncio.StreamWriter* 的方法), 1102
drive (*pathlib.PurePath* 的屬性), 447
drop_whitespace (*textwrap.TextWrapper* 的屬性), 175
dropwhile() (於 *itertools* 模組中), 414
dst() (*datetime.datetime* 的方法), 231
dst() (*datetime.time* 的方法), 238
dst() (*datetime.timezone* 的方法), 246
dst() (*datetime.tzinfo* 的方法), 240
DTDHandler (*xml.sax.handler* 中的類🄵), 1402
duck-typing（鴨子型🄵）, **2231**
dump() (*pickle.Pickler* 的方法), 508
dump() (*tracemalloc.Snapshot* 的方法), 1900
dump() (於 *ast* 模組中), 2127
dump() (於 *json* 模組中), 1322
dump() (於 *marshal* 模組中), 524
dump() (於 *pickle* 模組中), 507
dump() (於 *plistlib* 模組中), 648
dump() (於 *xml.etree.ElementTree* 模組中), 1373

dump_c_stack() (於 *faulthandler* 模組中), 1865
dump_stats() (*profile.Profile* 的方法), 1881
dump_stats() (*pstats.Stats* 的方法), 1882
dump_traceback() (於 *faulthandler* 模組中), 1865
dump_traceback_later() (於 *faulthandler* 模組中), 1866
dumps() (於 *json* 模組中), 1323
dumps() (於 *marshal* 模組中), 525
dumps() (於 *pickle* 模組中), 507
dumps() (於 *plistlib* 模組中), 648
dumps() (於 *xmlrpc.client* 模組中), 1536
dunder（雙底🄵）, **2232**
dup() (*socket.socket* 的方法), 1192
dup() (於 *os* 模組中), 678
dup2() (於 *os* 模組中), 678
DuplicateOptionError, 643
DuplicateSectionError, 643
--durations
    unittest 命令列選項, 1749
dwFlags (*subprocess.STARTUPINFO* 的屬性), 1055
DynamicClassAttribute() (於 *types* 模組中), 309

E

-e
    calendar 命令列選項, 262
    compileall 命令列選項, 2148
    idle 命令列選項, 1624
    mimetypes 命令列選項, 1351
    tarfile 命令列選項, 616
    tokenize 命令列選項, 2141
    zipfile 命令列選項, 601
e (於 *cmath* 模組中), 352
e (於 *math* 模組中), 348
E2BIG (於 *errno* 模組中), 809
EACCES (於 *errno* 模組中), 809
EADDRINUSE (於 *errno* 模組中), 813
EADDRNOTAVAIL (於 *errno* 模組中), 813
EADV (於 *errno* 模組中), 812
EAFNOSUPPORT (於 *errno* 模組中), 813
EAFP, **2232**
EAGAIN (於 *errno* 模組中), 809
eager_task_factory() (於 *asyncio* 模組中), 1087
EALREADY (於 *errno* 模組中), 814
east_asian_width() (於 *unicodedata* 模組中), 178
EAUTH (於 *errno* 模組中), 815
EBADARCH (於 *errno* 模組中), 815
EBADE (於 *errno* 模組中), 811
EBADEXEC (於 *errno* 模組中), 815
EBADF (於 *errno* 模組中), 809
EBADFD (於 *errno* 模組中), 812
EBADMACHO (於 *errno* 模組中), 815
EBADMSG (於 *errno* 模組中), 812
EBADR (於 *errno* 模組中), 811
EBADRPC (於 *errno* 模組中), 816
EBADRQC (於 *errno* 模組中), 811
EBADSLT (於 *errno* 模組中), 811
EBFONT (於 *errno* 模組中), 811
EBUSY (於 *errno* 模組中), 809
ECANCELED (於 *errno* 模組中), 816
ECHILD (於 *errno* 模組中), 809
echo() (於 *curses* 模組中), 930
echochar() (*curses.window* 的方法), 938
ECHRNG (於 *errno* 模組中), 810
ECOMM (於 *errno* 模組中), 812
ECONNABORTED (於 *errno* 模組中), 813
ECONNREFUSED (於 *errno* 模組中), 814
ECONNRESET (於 *errno* 模組中), 813
EDEADLK (於 *errno* 模組中), 810
EDEADLOCK (於 *errno* 模組中), 811
EDESTADDRREQ (於 *errno* 模組中), 813
EDEVERR (於 *errno* 模組中), 815
edit() (*curses.textpad.Textbox* 的方法), 953
EDOM (於 *errno* 模組中), 810
EDOTDOT (於 *errno* 模組中), 812
EDQUOT (於 *errno* 模組中), 814
EEXIST (於 *errno* 模組中), 809
EFAULT (於 *errno* 模組中), 809
EFBIG (於 *errno* 模組中), 810
EFD_CLOEXEC (於 *os* 模組中), 712
EFD_NONBLOCK (於 *os* 模組中), 712
EFD_SEMAPHORE (於 *os* 模組中), 712
effective() (於 *bdb* 模組中), 1864
EFTYPE (於 *errno* 模組中), 815
ehlo() (*smtplib.SMTP* 的方法), 1493
ehlo_or_helo_if_needed() (*smtplib.SMTP* 的方法), 1493
EHOSTDOWN (於 *errno* 模組中), 814
EHOSTUNREACH (於 *errno* 模組中), 814
EHWPOISON (於 *errno* 模組中), 814
EIDRM (於 *errno* 模組中), 810
EILSEQ (於 *errno* 模組中), 812
EINPROGRESS (於 *errno* 模組中), 814
EINTR (於 *errno* 模組中), 809
EINVAL (於 *errno* 模組中), 809
EIO (於 *errno* 模組中), 809
EISCONN (於 *errno* 模組中), 813
EISDIR (於 *errno* 模組中), 809
EISNAM (於 *errno* 模組中), 814
EJECT (*enum.FlagBoundary* 的屬性), 331
EKEYEXPIRED (於 *errno* 模組中), 815
EKEYREJECTED (於 *errno* 模組中), 815
EKEYREVOKED (於 *errno* 模組中), 815
EL2HLT (於 *errno* 模組中), 811
EL2NSYNC (於 *errno* 模組中), 810
EL3HLT (於 *errno* 模組中), 811
EL3RST (於 *errno* 模組中), 811
Element (*xml.etree.ElementTree* 中的類🄵), 1377
element_create() (*tkinter.ttk.Style* 的方法), 1614
element_names() (*tkinter.ttk.Style* 的方法), 1615
element_options() (*tkinter.ttk.Style* 的方法), 1615
ElementDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
elements() (*collections.Counter* 的方法), 267
ElementTree (*xml.etree.ElementTree* 中的類🄵), 1379
ELIBACC (於 *errno* 模組中), 812
ELIBBAD (於 *errno* 模組中), 812

ELIBEXEC (於 *errno* 模組中), 812
ELIBMAX (於 *errno* 模組中), 812
ELIBSCN (於 *errno* 模組中), 812
Ellipsis (🄵建變數), 36
ELLIPSIS (於 *doctest* 模組中), 1731
ELLIPSIS (於 *token* 模組中), 2138
EllipsisType (於 *types* 模組中), 306
ELNRNG (於 *errno* 模組中), 811
ELOCKUNMAPPED (於 *errno* 模組中), 815
ELOOP (於 *errno* 模組中), 810
EM (於 *curses.ascii* 模組中), 956
email
    module, 1263
email.charset
    module, 1313
email.contentmanager
    module, 1292
email.encoders
    module, 1315
email.errors
    module, 1285
email.generator
    module, 1275
email.header
    module, 1311
email.headerregistry
    module, 1286
email.iterators
    module, 1318
email.message
    module, 1264
EmailMessage (*email.message* 中的類🄵), 1264
email.mime
    module, 1308
email.mime.application
    module, 1309
email.mime.audio
    module, 1309
email.mime.base
    module, 1308
email.mime.image
    module, 1310
email.mime.message
    module, 1310
email.mime.multipart
    module, 1309
email.mime.nonmultipart
    module, 1309
email.mime.text
    module, 1310
email.parser
    module, 1272
email.policy
    module, 1278
EmailPolicy (*email.policy* 中的類🄵), 1282
email.utils
    module, 1316
Emax (*decimal.Context* 的屬性), 366
EMEDIUMTYPE (於 *errno* 模組中), 814
EMFILE (於 *errno* 模組中), 809
Emin (*decimal.Context* 的屬性), 366
emit() (*logging.FileHandler* 的方法), 791
emit() (*logging.Handler* 的方法), 767
emit() (*logging.handlers.BufferingHandler* 的方法), 800
emit() (*logging.handlers.DatagramHandler* 的方法), 797
emit() (*logging.handlers.HTTPHandler* 的方法), 801
emit() (*logging.handlers.NTEventLogHandler* 的方法), 799
emit() (*logging.handlers.QueueHandler* 的方法), 802
emit() (*logging.handlers.RotatingFileHandler* 的方法), 794
emit() (*logging.handlers.SMTPHandler* 的方法), 800
emit() (*logging.handlers.SocketHandler* 的方法), 796
emit() (*logging.handlers.SysLogHandler* 的方法), 798
emit() (*logging.handlers.TimedRotatingFileHandler* 的方法), 795
emit() (*logging.handlers.WatchedFileHandler* 的方法), 792
emit() (*logging.NullHandler* 的方法), 791
emit() (*logging.StreamHandler* 的方法), 790
EMLINK (於 *errno* 模組中), 810
Empty, 1065
empty (*inspect.Parameter* 的屬性), 2024
empty (*inspect.Signature* 的屬性), 2023
empty() (*asyncio.Queue* 的方法), 1116
empty() (*multiprocessing.Queue* 的方法), 994
empty() (*multiprocessing.SimpleQueue* 的方法), 995
empty() (*queue.Queue* 的方法), 1065
empty() (*queue.SimpleQueue* 的方法), 1067
empty() (*sched.scheduler* 的方法), 1064
EMPTY_NAMESPACE (於 *xml.dom* 模組中), 1385
emptyline() (*cmd.Cmd* 的方法), 960
emscripten_version (*sys._emscripten_info* 的屬性), 1923
EMSGSIZE (於 *errno* 模組中), 813
EMULTIHOP (於 *errno* 模組中), 812
enable (*pdb command*), 1873
enable() (*bdb.Breakpoint* 的方法), 1859
enable() (*imaplib.IMAP4* 的方法), 1485
enable() (*profile.Profile* 的方法), 1881
enable() (於 *faulthandler* 模組中), 1866
enable() (於 *gc* 模組中), 2012
enable_callback_tracebacks() (於 *sqlite3* 模組中), 535
enable_interspersed_args() (*optparse.OptionParser* 的方法), 916
enable_load_extension() (*sqlite3.Connection* 的方法), 543
enable_long_distance_matching (*compression.zstd.CompressionParameter* 的屬性), 570
enable_traversal() (*tkinter.ttk.Notebook* 的方法), 1605
ENABLE_USER_SITE (於 *site* 模組中), 2045

enabled (*bdb.Breakpoint* 的屬性), 1860
EnableReflectionKey() (於 *winreg* 模組中), 2185
ENAMETOOLONG (於 *errno* 模組中), 810
ENAVAIL (於 *errno* 模組中), 814
enclose() (*curses.window* 的方法), 938
encode (*codecs.CodecInfo* 的屬性), 194
encode() (*codecs.Codec* 的方法), 199
encode() (*codecs.IncrementalEncoder* 的方法), 200
encode() (*email.header.Header* 的方法), 1312
encode() (*json.JSONEncoder* 的方法), 1326
encode() (*str* 的方法), 55
encode() (於 *base64* 模組中), 1355
encode() (於 *codecs* 模組中), 194
encode() (於 *quopri* 模組中), 1358
encode() (*xmlrpc.client.Binary* 的方法), 1534
encode() (*xmlrpc.client.DateTime* 的方法), 1533
encode_7or8bit() (於 *email.encoders* 模組中), 1316
encode_base64() (於 *email.encoders* 模組中), 1316
encode_noop() (於 *email.encoders* 模組中), 1316
encode_quopri() (於 *email.encoders* 模組中), 1315
encode_rfc2231() (於 *email.utils* 模組中), 1318
encodebytes() (於 *base64* 模組中), 1355
EncodedFile() (於 *codecs* 模組中), 196
encodePriority() (*logging.handlers.SysLogHandler* 的方法), 798
encodestring() (於 *quopri* 模組中), 1358
encode（編碼）
    Codecs, 194
--encoding
    calendar 命令列選項, 262
encoding (*curses.window* 的屬性), 938
encoding (*io.TextIOBase* 的屬性), 743
encoding (*UnicodeError* 的屬性), 115
ENCODING (於 *tarfile* 模組中), 605
ENCODING (於 *token* 模組中), 2135
encodings
    module, 209
encodings_map (*mimetypes.MimeTypes* 的屬性), 1350
encodings_map (於 *mimetypes* 模組中), 1349
encodings.idna
    module, 210
encodings.mbcs
    module, 211
encodings.utf_8_sig
    module, 211
EncodingWarning, 118
encoding（編碼）
    base64, 1352
    quoted-printable（可列印字元）, 1358
end (*UnicodeError* 的屬性), 115
end() (*re.Match* 的方法), 155
end() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
END_ASYNC_FOR (*opcode*), 2160
end_col_offset (*ast.AST* 的屬性), 2096
end_colno (*traceback.FrameSummary* 的屬性), 2006
end_fill() (於 *turtle* 模組中), 1645
END_FOR (*opcode*), 2157
end_headers() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
end_lineno (*ast.AST* 的屬性), 2096
end_lineno (*SyntaxError* 的屬性), 114
end_lineno (*traceback.FrameSummary* 的屬性), 2006
end_lineno (*traceback.TracebackException* 的屬性), 2004
end_ns() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
end_offset (*SyntaxError* 的屬性), 114
end_offset (*traceback.TracebackException* 的屬性), 2004
end_poly() (於 *turtle* 模組中), 1650
END_SEND (*opcode*), 2157
endCDATA() (*xml.sax.handler.LexicalHandler* 的方法), 1406
EndCdataSectionHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
EndDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
endDocument() (*xml.sax.handler.ContentHandler* 的方法), 1404
endDTD() (*xml.sax.handler.LexicalHandler* 的方法), 1406
endElement() (*xml.sax.handler.ContentHandler* 的方法), 1404
EndElementHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
endElementNS() (*xml.sax.handler.ContentHandler* 的方法), 1404
endheaders() (*http.client.HTTPConnection* 的方法), 1471
ENDMARKER (於 *token* 模組中), 2135
EndNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
endpos (*re.Match* 的屬性), 155
endPrefixMapping() (*xml.sax.handler.ContentHandler* 的方法), 1404
endswith() (*bytearray* 的方法), 71
endswith() (*bytes* 的方法), 71
endswith() (*str* 的方法), 55
endwin() (於 *curses* 模組中), 930
ENEEDAUTH (於 *errno* 模組中), 815
ENETDOWN (於 *errno* 模組中), 813
ENETRESET (於 *errno* 模組中), 813
ENETUNREACH (於 *errno* 模組中), 813
ENFILE (於 *errno* 模組中), 809
ENOANO (於 *errno* 模組中), 811
ENOATTR (於 *errno* 模組中), 815
ENOBUFS (於 *errno* 模組中), 813
ENOCSI (於 *errno* 模組中), 811
ENODATA (於 *errno* 模組中), 811
ENODEV (於 *errno* 模組中), 809

ENOENT (於 *errno* 模組中), 808
ENOEXEC (於 *errno* 模組中), 809
ENOKEY (於 *errno* 模組中), 815
ENOLCK (於 *errno* 模組中), 810
ENOLINK (於 *errno* 模組中), 812
ENOMEDIUM (於 *errno* 模組中), 814
ENOMEM (於 *errno* 模組中), 809
ENOMSG (於 *errno* 模組中), 810
ENONET (於 *errno* 模組中), 811
ENOPKG (於 *errno* 模組中), 811
ENOPOLICY (於 *errno* 模組中), 815
ENOPROTOOPT (於 *errno* 模組中), 813
ENOSPC (於 *errno* 模組中), 810
ENOSR (於 *errno* 模組中), 811
ENOSTR (於 *errno* 模組中), 811
ENOSYS (於 *errno* 模組中), 810
ENOTACTIVE (於 *errno* 模組中), 815
ENOTBLK (於 *errno* 模組中), 809
ENOTCAPABLE (於 *errno* 模組中), 816
ENOTCONN (於 *errno* 模組中), 814
ENOTDIR (於 *errno* 模組中), 809
ENOTEMPTY (於 *errno* 模組中), 810
ENOTNAM (於 *errno* 模組中), 814
ENOTRECOVERABLE (於 *errno* 模組中), 816
ENOTSOCK (於 *errno* 模組中), 813
ENOTSUP (於 *errno* 模組中), 813
ENOTTY (於 *errno* 模組中), 810
ENOTUNIQ (於 *errno* 模組中), 812
ENQ (於 *curses.ascii* 模組中), 955
enqueue() (*logging.handlers.QueueHandler* 的方法), 802
enqueue_sentinel() (*logging.handlers.QueueListener* 的方法), 804
ensure_directories() (*venv.EnvBuilder* 的方法), 1909
ensure_future() (於 *asyncio* 模組中), 1144
ensurepip
    module, 1903
ensurepip 命令列選項
    --altinstall, 1904
    --default-pip, 1904
    --root, 1904
    --user, 1904
enter() (*sched.scheduler* 的方法), 1063
enter_async_context() (*contextlib.AsyncExitStack* 的方法), 1988
enter_context() (*contextlib.ExitStack* 的方法), 1987
enterabs() (*sched.scheduler* 的方法), 1063
enterAsyncContext() (*unittest.IsolatedAsyncioTestCase* 的方法), 1766
enterClassContext() (*unittest.TestCase* 的類🄵方法), 1765
enterContext() (*unittest.TestCase* 的方法), 1764
enterModuleContext() (於 *unittest* 模組中), 1775
entities (*xml.dom.DocumentType* 的屬性), 1389
EntityDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
entitydefs (於 *html.entities* 模組中), 1364
EntityResolver (*xml.sax.handler* 中的類🄵), 1402
entry_points() (於 *importlib.metadata* 模組中), 2085
EntryPoint (*importlib.metadata* 中的類🄵), 2085
EntryPoints (*importlib.metadata* 中的類🄵), 2085
enum
    module, 319
Enum (*enum* 中的類🄵), 322
enum_certificates() (於 *ssl* 模組中), 1207
enum_crls() (於 *ssl* 模組中), 1207
EnumCheck (*enum* 中的類🄵), 329
EnumDict (*enum* 中的類🄵), 331
enumerate()
    built-in function, 13
enumerate() (於 *threading* 模組中), 967
EnumKey() (於 *winreg* 模組中), 2181
EnumType (*enum* 中的類🄵), 321
EnumValue() (於 *winreg* 模組中), 2182
ENV_DIR
    venv 命令列選項, 1906
EnvBuilder (*venv* 中的類🄵), 1908
environ (於 *os* 模組中), 669
environ (於 *posix* 模組中), 2197
environb (於 *os* 模組中), 670
environment variables (環境變數)
    deleting (🄵除), 676
    setting (設定), 673
EnvironmentError, 116
Environments (環境)
    virtual (🄵擬), 1905
EnvironmentVarGuard (*test.support.os_helper* 中的類🄵), 1849
ENXIO (於 *errno* 模組中), 809
eof (*bz2.BZ2Decompressor* 的屬性), 584
eof (*compression.zstd.ZstdDecompressor* 的屬性), 567
eof (*lzma.LZMADecompressor* 的屬性), 588
eof (*shlex.shlex* 的屬性), 2194
eof (*ssl.MemoryBIO* 的屬性), 1234
eof (*zlib.Decompress* 的屬性), 576
eof_received() (*asyncio.BufferedProtocol* 的方法), 1153
eof_received() (*asyncio.Protocol* 的方法), 1153
EOFError, 111
EOPNOTSUPP (於 *errno* 模組中), 813
EOT (於 *curses.ascii* 模組中), 955
EOVERFLOW (於 *errno* 模組中), 812
EOWNERDEAD (於 *errno* 模組中), 816
EPERM (於 *errno* 模組中), 808
EPFNOSUPPORT (於 *errno* 模組中), 813
epilogue (*email.message.EmailMessage* 的屬性), 1271
epilogue (*email.message.Message* 的屬性), 1308
EPIPE (於 *errno* 模組中), 810
epoch (紀元), 748
epoll() (於 *select* 模組中), 1237
EpollSelector (*selectors* 中的類🄵), 1246
EPROCLIM (於 *errno* 模組中), 816
EPROCUNAVAIL (於 *errno* 模組中), 816

EPROGMISMATCH (於 *errno* 模組中), 816
EPROGUNAVAIL (於 *errno* 模組中), 816
EPROTO (於 *errno* 模組中), 812
EPROTONOSUPPORT (於 *errno* 模組中), 813
EPROTOTYPE (於 *errno* 模組中), 813
epsilon (*sys.float_info* 的屬性), 1928
EPWROFF (於 *errno* 模組中), 816
Eq (*ast* 中的類F), 2103
eq() (於 *operator* 模組中), 435
EQEQUAL (於 *token* 模組中), 2137
EQFULL (於 *errno* 模組中), 814
EQUAL (於 *token* 模組中), 2136
ERA (於 *locale* 模組中), 1572
ERA_D_FMT (於 *locale* 模組中), 1573
ERA_D_T_FMT (於 *locale* 模組中), 1572
ERA_T_FMT (於 *locale* 模組中), 1573
ERANGE (於 *errno* 模組中), 810
erase() (*curses.window* 的方法), 938
erasechar() (於 *curses* 模組中), 930
EREMCHG (於 *errno* 模組中), 812
EREMOTE (於 *errno* 模組中), 811
EREMOTEIO (於 *errno* 模組中), 814
ERESTART (於 *errno* 模組中), 812
erf() (於 *math* 模組中), 347
erfc() (於 *math* 模組中), 348
ERFKILL (於 *errno* 模組中), 815
EROFS (於 *errno* 模組中), 810
ERPCMISMATCH (於 *errno* 模組中), 816
ERR (於 *curses* 模組中), 942
errcheck (*ctypes._CFuncPtr* 的屬性), 839
errcode (*xmlrpc.client.ProtocolError* 的屬性), 1535
errmsg (*xmlrpc.client.ProtocolError* 的屬性), 1535
errno
    module, 808
    module（模組）, 112
errno (*OSError* 的屬性), 112
Error, 310, 498, 552, 623, 643, 1346, 1358, 1424, 1557, 1569
error, 187, 526, 528530, 574, 667, 929, 1071, 1177, 1237, 1411, 2206, 2216
ERROR (於 *logging* 模組中), 766
ERROR (於 *tkinter.messagebox* 模組中), 1597
error handler's name（錯誤處理器名稱）
    backslashreplace, 197
    ignore, 197
    namereplace, 197
    replace, 197
    strict, 197
    surrogateescape, 197
    surrogatepass, 198
    xmlcharrefreplace, 197
error() (*argparse.ArgumentParser* 的方法), 884
error() (*logging.Logger* 的方法), 764
error() (*urllib.request.OpenerDirector* 的方法), 1443
error() (於 *logging* 模組中), 774
error() (*xml.sax.handler.ErrorHandler* 的方法), 1406
error_body (*wsgiref.handlers.BaseHandler* 的屬性), 1433
error_content_type (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
error_headers (*wsgiref.handlers.BaseHandler* 的屬性), 1432
error_leader() (*shlex.shlex* 的方法), 2193
error_message_format (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
error_output() (*wsgiref.handlers.BaseHandler* 的方法), 1432
error_perm, 1479
error_proto, 1479, 1480
error_received() (*asyncio.DatagramProtocol* 的方法), 1154
error_reply, 1479
error_status (*wsgiref.handlers.BaseHandler* 的屬性), 1432
error_temp, 1479
ErrorByteIndex (*xml.parsers.expat.xmlparser* 的屬性), 1414
errorcode (於 *errno* 模組中), 808
ErrorCode (*xml.parsers.expat.xmlparser* 的屬性), 1414
ErrorColumnNumber (*xml.parsers.expat.xmlparser* 的屬性), 1414
ErrorHandler (*xml.sax.handler* 中的類F), 1402
errorlevel (*tarfile.TarFile* 的屬性), 608
ErrorLineNumber (*xml.parsers.expat.xmlparser* 的屬性), 1414
errors (*io.TextIOBase* 的屬性), 743
errors (*unittest.TestLoader* 的屬性), 1767
errors (*unittest.TestResult* 的屬性), 1770
ErrorStream (*wsgiref.types* 中的類F), 1433
ErrorString() (於 *xml.parsers.expat* 模組中), 1412
Errors（錯誤）
    logging（日F）, 759
ERRORTOKEN (於 *token* 模組中), 2135
ESC (於 *curses.ascii* 模組中), 956
escape (*shlex.shlex* 的屬性), 2194
escape() (於 *glob* 模組中), 489
escape() (於 *html* 模組中), 1359
escape() (於 *re* 模組中), 151
escape() (於 *xml.sax.saxutils* 模組中), 1406
escapechar (*csv.Dialect* 的屬性), 623
escapedquotes (*shlex.shlex* 的屬性), 2194
ESHLIBVERS (於 *errno* 模組中), 816
ESHUTDOWN (於 *errno* 模組中), 814
ESOCKTNOSUPPORT (於 *errno* 模組中), 813
ESPIPE (於 *errno* 模組中), 810
ESRCH (於 *errno* 模組中), 808
ESRMNT (於 *errno* 模組中), 812
ESTALE (於 *errno* 模組中), 814
ESTRPIPE (於 *errno* 模組中), 812
ETB (於 *curses.ascii* 模組中), 956
ETH_P_ALL (於 *socket* 模組中), 1180
ETHERTYPE_ARP (於 *socket* 模組中), 1183
ETHERTYPE_IP (於 *socket* 模組中), 1183
ETHERTYPE_IPV6 (於 *socket* 模組中), 1183

ETHERTYPE_VLAN (於 *socket* 模組中), 1183
ETIME (於 *errno* 模組中), 811
ETIMEDOUT (於 *errno* 模組中), 814
Etiny() (*decimal.Context* 的方法), 367
ETOOMANYREFS (於 *errno* 模組中), 814
Etop() (*decimal.Context* 的方法), 367
ETX (於 *curses.ascii* 模組中), 955
ETXTBSY (於 *errno* 模組中), 810
EUCLEAN (於 *errno* 模組中), 814
EUNATCH (於 *errno* 模組中), 811
EUSERS (於 *errno* 模組中), 812
eval
    built-in function(🄵建函式), 104, 312, 313
eval()
    built-in function, 14
evaluate function（求值函式）, **2232**
evaluate() (*annotationlib.ForwardRef* 的方法), 2037
evaluate_bound() (*typing.TypeVar* 的方法), 1692
evaluate_constraints() (*typing.TypeVar* 的方法), 1692
evaluate_default() (*typing.ParamSpec* 的方法), 1696
evaluate_default() (*typing.TypeVar* 的方法), 1693
evaluate_default() (*typing.TypeVarTuple* 的方法), 1694
evaluate_forward_ref() (於 *typing* 模組中), 1713
evaluate_value() (*typing.TypeAliasType* 的方法), 1698
Event (*asyncio* 中的類🄵), 1106
Event (*multiprocessing* 中的類🄵), 1000
Event (*threading* 中的類🄵), 979
event scheduling（事件排程）, 1063
Event() (*multiprocessing.managers.SyncManager* 的方法), 1006
EVENT_READ (於 *selectors* 模組中), 1244
EVENT_WRITE (於 *selectors* 模組中), 1244
eventfd() (於 *os* 模組中), 711
eventfd_read() (於 *os* 模組中), 711
eventfd_write() (於 *os* 模組中), 711
EventLoop (*asyncio* 中的類🄵), 1141
events (*selectors.SelectorKey* 的屬性), 1245
EWOULDBLOCK (於 *errno* 模組中), 810
EX_CANTCREAT (於 *os* 模組中), 718
EX_CONFIG (於 *os* 模組中), 718
EX_DATAERR (於 *os* 模組中), 717
EX_IOERR (於 *os* 模組中), 718
EX_NOHOST (於 *os* 模組中), 718
EX_NOINPUT (於 *os* 模組中), 717
EX_NOPERM (於 *os* 模組中), 718
EX_NOTFOUND (於 *os* 模組中), 718
EX_NOUSER (於 *os* 模組中), 718
EX_OK (於 *os* 模組中), 717
EX_OSERR (於 *os* 模組中), 718
EX_OSFILE (於 *os* 模組中), 718
EX_PROTOCOL (於 *os* 模組中), 718
EX_SOFTWARE (於 *os* 模組中), 718
EX_TEMPFAIL (於 *os* 模組中), 718
EX_UNAVAILABLE (於 *os* 模組中), 718
EX_USAGE (於 *os* 模組中), 717
--exact
    tokenize 命令列選項, 2141
EXACT_TOKEN_TYPES (於 *token* 模組中), 2138
Example (*doctest* 中的類🄵), 1739
example (*doctest.DocTestFailure* 的屬性), 1745
example (*doctest.UnexpectedException* 的屬性), 1745
examples (*doctest.DocTest* 的屬性), 1738
exc_info (*doctest.UnexpectedException* 的屬性), 1745
exc_info() (於 *sys* 模組中), 1924
exc_msg (*doctest.Example* 的屬性), 1739
exc_type (*traceback.TracebackException* 的屬性), 2004
exc_type_str (*traceback.TracebackException* 的屬性), 2004
excel (*csv* 中的類🄵), 622
excel_tab (*csv* 中的類🄵), 622
except
    statement（陳述式）, 109
ExceptHandler (*ast* 中的類🄵), 2113
excepthook() (於 *sys* 模組中), 1923
excepthook() (於 *threading* 模組中), 967
Exception, 110
EXCEPTION (於 *_tkinter* 模組中), 1591
exception() (*asyncio.Future* 的方法), 1145
exception() (*asyncio.Task* 的方法), 1095
exception() (*concurrent.futures.Future* 的方法), 1037
exception() (*logging.Logger* 的方法), 764
exception() (於 *logging* 模組中), 774
exception() (於 *sys* 模組中), 1924
EXCEPTION_HANDLED (*monitoring event*), 1947
ExceptionGroup, 118
exceptions (*BaseExceptionGroup* 的屬性), 119
exceptions (*pdb command*), 1877
exceptions (*traceback.TracebackException* 的屬性), 2003
exception（例外）
    chaining, 109
excinfo (*concurrent.interpreters.ExecutionFailed* 的屬性), 1043
EXCLAMATION (於 *token* 模組中), 2138
EXDEV (於 *errno* 模組中), 809
exec
    built-in function（🄵建函式）, 15, 104
exec()
    built-in function, 15
exec() (*concurrent.interpreters.Interpreter* 的方法), 1042
exec_module() (*importlib.abc.InspectLoader* 的方法), 2064
exec_module() (*importlib.abc.Loader* 的方法), 2063
exec_module() (*importlib.abc.SourceLoader* 的方法), 2066
exec_module() (*importlib.machinery.ExtensionFileLoader* 的方法), 2071

exec_module() (*zipimport.zipimporter* 的方法), 2052
exec_prefix (於 *sys* 模組中), 1924
execl() (於 *os* 模組中), 716
execle() (於 *os* 模組中), 716
execlp() (於 *os* 模組中), 716
execlpe() (於 *os* 模組中), 716
executable (於 *sys* 模組中), 1924
Executable Zip Files (可執行的 Zip 檔案), 1914
execute() (*sqlite3.Connection* 的方法), 539
execute() (*sqlite3.Cursor* 的方法), 548
executemany() (*sqlite3.Connection* 的方法), 539
executemany() (*sqlite3.Cursor* 的方法), 548
executescript() (*sqlite3.Connection* 的方法), 539
executescript() (*sqlite3.Cursor* 的方法), 549
ExecutionFailed, 1039, 1043
ExecutionLoader (*importlib.abc* 中的類🄵), 2064
Executor (*concurrent.futures* 中的類🄵), 1032
execv() (於 *os* 模組中), 716
execve() (於 *os* 模組中), 716
execvp() (於 *os* 模組中), 716
execvpe() (於 *os* 模組中), 716
EXFULL (於 *errno* 模組中), 811
exists() (*pathlib.Path* 的方法), 457
exists() (*pathlib.types.PathInfo* 的方法), 468
exists() (*tkinter.ttk.Treeview* 的方法), 1610
exists() (於 *os.path* 模組中), 470
exists() (*zipfile.Path* 的方法), 597
exit (🄵建變數), 36
exit() (*argparse.ArgumentParser* 的方法), 884
exit() (於 *_thread* 模組中), 1072
exit() (於 *sys* 模組中), 1924
exitcode (*multiprocessing.Process* 的屬性), 990
exitonclick() (於 *turtle* 模組中), 1658
ExitStack (*contextlib* 中的類🄵), 1987
exp() (*decimal.Context* 的方法), 368
exp() (*decimal.Decimal* 的方法), 360
exp() (於 *cmath* 模組中), 351
exp() (於 *math* 模組中), 345
exp2() (於 *math* 模組中), 345
expand() (*re.Match* 的方法), 153
expand_tabs (*textwrap.TextWrapper* 的屬性), 175
ExpandEnvironmentStrings() (於 *winreg* 模組中), 2182
expandNode() (*xml.dom.pulldom.DOMEventStream* 的方法), 1399
expandtabs() (*bytearray* 的方法), 76
expandtabs() (*bytes* 的方法), 76
expandtabs() (*str* 的方法), 55
expanduser() (*pathlib.Path* 的方法), 456
expanduser() (於 *os.path* 模組中), 470
expandvars() (於 *os.path* 模組中), 470
Expat, 1411
ExpatError, 1411
expected (*asyncio.IncompleteReadError* 的屬性), 1118
expectedFailure() (於 *unittest* 模組中), 1754
expectedFailures (*unittest.TestResult* 的屬性), 1770
expired() (*asyncio.Timeout* 的方法), 1089
expires (*http.cookiejar.Cookie* 的屬性), 1529
expires (*http.cookies.Morsel* 的屬性), 1520
exploded (*ipaddress.IPv4Address* 的屬性), 1545
exploded (*ipaddress.IPv4Network* 的屬性), 1550
exploded (*ipaddress.IPv6Address* 的屬性), 1547
exploded (*ipaddress.IPv6Network* 的屬性), 1553
expm1() (於 *math* 模組中), 345
expovariate() (於 *random* 模組中), 387
Expr (*ast* 中的類🄵), 2101
Expression (*ast* 中的類🄵), 2097
expression (*string.templatelib.Interpolation* 的屬性), 138
expression (運算式), **2232**
expunge() (*imaplib.IMAP4* 的方法), 1485
extend() (*array.array* 的方法), 295
extend() (*collections.deque* 的方法), 269
extend() (*sequence* 的方法), 49
extend() (*xml.etree.ElementTree.Element* 的方法), 1378
extend_path() (於 *pkgutil* 模組中), 2053
EXTENDED_ARG (*opcode*), 2169
ExtendedContext (於 *decimal* 模組中), 365
ExtendedInterpolation (*configparser* 中的類🄵), 631
extendleft() (*collections.deque* 的方法), 269
--extension
    mimetypes 命令列選項, 1351
extension module (擴充模組), **2232**
EXTENSION_SUFFIXES (於 *importlib.machinery* 模組中), 2068
ExtensionFileLoader (*importlib.machinery* 中的類🄵), 2071
extensions_map (*http.server.SimpleHTTPRequestHandler* 的屬性), 1515
External Data Representation (外部資料表現), 506
external_attr (*zipfile.ZipInfo* 的屬性), 600
ExternalClashError, 1347
ExternalEntityParserCreate() (*xml.parsers.expat.xmlparser* 的方法), 1413
ExternalEntityRefHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
extra (*zipfile.ZipInfo* 的屬性), 600
--extract
    tarfile 命令列選項, 616
    zipfile 命令列選項, 601
extract() (*tarfile.TarFile* 的方法), 608
extract() (*traceback.StackSummary* 的類🄵方法), 2005
extract() (*zipfile.ZipFile* 的方法), 594
extract_cookies() (*http.cookiejar.CookieJar* 的方法), 1524
extract_stack() (於 *traceback* 模組中), 2002
extract_tb() (於 *traceback* 模組中), 2001
extract_version (*zipfile.ZipInfo* 的屬性), 600

extractall() (*tarfile.TarFile* 的方法), 607
extractall() (*zipfile.ZipFile* 的方法), 595
ExtractError, 604
extractfile() (*tarfile.TarFile* 的方法), 608
extraction_filter (*tarfile.TarFile* 的屬性), 609
extsep (於 *os* 模組中), 732

F

-f
    calendar 命令列選項, 263
    compileall 命令列選項, 2147
    doctest 命令列選項, 1727
    random 命令列選項, 393
    trace 命令列選項, 1891
    unittest 命令列選項, 1749
f-strings (f 字串), **2232**
f-string (f 字串), **2232**
F_CONTIGUOUS (*inspect.BufferFlags* 的屬性), 2034
f_contiguous (*memoryview* 的屬性), 89
F_LOCK (於 *os* 模組中), 680
F_OK (於 *os* 模組中), 691
F_TEST (於 *os* 模組中), 680
F_TLOCK (於 *os* 模組中), 680
F_ULOCK (於 *os* 模組中), 680
fabs() (於 *math* 模組中), 342
factorial() (於 *math* 模組中), 341
factory() (*importlib.util.LazyLoader* 的類🄵方法), 2076
fail() (*unittest.TestCase* 的方法), 1763
FAIL_FAST (於 *doctest* 模組中), 1732
failed (*doctest.TestResults* 的屬性), 1741
--failfast
    unittest 命令列選項, 1749
--fail-fast
    doctest 命令列選項, 1727
failfast (*unittest.TestResult* 的屬性), 1771
failureException (*unittest.TestCase* 的屬性), 1763
failures (*doctest.DocTestRunner* 的屬性), 1742
failures (*unittest.TestResult* 的屬性), 1770
FakePath (*test.support.os_helper* 中的類🄵), 1849
False, 37, 45
false, 37
False (🄵建變數), 35
False (🄵建物件), 37
families() (於 *tkinter.font* 模組中), 1592
family (*socket.socket* 的屬性), 1198
--fast
    gzip 命令列選項, 581
fast (*compression.zstd.Strategy* 的屬性), 572
fast (*pickle.Pickler* 的屬性), 509
fatalError() (*xml.sax.handler.ErrorHandler* 的方法), 1406
Fault (*xmlrpc.client* 中的類🄵), 1534
faultCode (*xmlrpc.client.Fault* 的屬性), 1534
faulthandler
    module, 1864
faultString (*xmlrpc.client.Fault* 的屬性), 1534
fchdir() (於 *os* 模組中), 693
fchmod() (於 *os* 模組中), 678
fchown() (於 *os* 模組中), 678
fcntl
    module, 2203
fcntl() (於 *fcntl* 模組中), 2203
fd (*selectors.SelectorKey* 的屬性), 1244
fd() (於 *turtle* 模組中), 1635
fd_count() (於 *test.support.os_helper* 模組中), 1850
fdatasync() (於 *os* 模組中), 679
fdopen() (於 *os* 模組中), 677
feature_external_ges (於 *xml.sax.handler* 模組中), 1402
feature_external_pes (於 *xml.sax.handler* 模組中), 1402
feature_namespace_prefixes (於 *xml.sax.handler* 模組中), 1402
feature_namespaces (於 *xml.sax.handler* 模組中), 1402
feature_string_interning (於 *xml.sax.handler* 模組中), 1402
feature_validation (於 *xml.sax.handler* 模組中), 1402
--feature-version
    ast 命令列選項, 2129
FEBRUARY (於 *calendar* 模組中), 261
feed() (*email.parser.BytesFeedParser* 的方法), 1273
feed() (*html.parser.HTMLParser* 的方法), 1361
feed() (*xml.etree.ElementTree.XMLParser* 的方法), 1382
feed() (*xml.etree.ElementTree.XMLPullParser* 的方法), 1383
feed() (*xml.sax.xmlreader.IncrementalParser* 的方法), 1409
feed_eof() (*asyncio.StreamReader* 的方法), 1100
FeedParser (*email.parser* 中的類🄵), 1273
fetch() (*imaplib.IMAP4* 的方法), 1485
fetchall() (*sqlite3.Cursor* 的方法), 549
fetchmany() (*sqlite3.Cursor* 的方法), 549
fetchone() (*sqlite3.Cursor* 的方法), 549
FF (於 *curses.ascii* 模組中), 955
fflags (*select.kevent* 的屬性), 1242
Field (*dataclasses* 中的類🄵), 1973
field() (於 *dataclasses* 模組中), 1972
field_size_limit() (於 *csv* 模組中), 620
fieldnames (*csv.DictReader* 的屬性), 624
fields (*uuid.UUID* 的屬性), 1498
fields() (於 *dataclasses* 模組中), 1974
FIFOTYPE (於 *tarfile* 模組中), 605
file
    compileall 命令列選項, 2147
    gzip 命令列選項, 581
--file
    trace 命令列選項, 1891
file (*bdb.Breakpoint* 的屬性), 1860
file (*pyclbr.Class* 的屬性), 2145
file (*pyclbr.Function* 的屬性), 2144
file control (檔案控制)
    UNIX, 2203

file name（檔案名稱）
    temporary（臨時）, 483
file object（檔案物件）, **2232**
    io 模組, 734
    open() 🄵建函式, 22
file-like object（類檔案物件）, **2232**
FILE_ATTRIBUTE_ARCHIVE (於 *stat* 模組中), 480
FILE_ATTRIBUTE_COMPRESSED (於 *stat* 模組中), 480
FILE_ATTRIBUTE_DEVICE (於 *stat* 模組中), 480
FILE_ATTRIBUTE_DIRECTORY (於 *stat* 模組中), 480
FILE_ATTRIBUTE_ENCRYPTED (於 *stat* 模組中), 480
FILE_ATTRIBUTE_HIDDEN (於 *stat* 模組中), 480
FILE_ATTRIBUTE_INTEGRITY_STREAM (於 *stat* 模組中), 480
FILE_ATTRIBUTE_NO_SCRUB_DATA (於 *stat* 模組中), 480
FILE_ATTRIBUTE_NORMAL (於 *stat* 模組中), 480
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (於 *stat* 模組中), 480
FILE_ATTRIBUTE_OFFLINE (於 *stat* 模組中), 480
FILE_ATTRIBUTE_READONLY (於 *stat* 模組中), 480
FILE_ATTRIBUTE_REPARSE_POINT (於 *stat* 模組中), 480
FILE_ATTRIBUTE_SPARSE_FILE (於 *stat* 模組中), 480
FILE_ATTRIBUTE_SYSTEM (於 *stat* 模組中), 480
FILE_ATTRIBUTE_TEMPORARY (於 *stat* 模組中), 480
FILE_ATTRIBUTE_VIRTUAL (於 *stat* 模組中), 480
file_digest() (於 *hashlib* 模組中), 654
file_open() (*urllib.request.FileHandler* 的方法), 1447
file_size (*zipfile.ZipInfo* 的屬性), 601
filecmp
    module, 481
fileConfig() (於 *logging.config* 模組中), 779
FileCookieJar (*http.cookiejar* 中的類🄵), 1522
FileDialog (*tkinter.filedialog* 中的類🄵), 1594
FileExistsError, 117
FileFinder (*importlib.machinery* 中的類🄵), 2069
FileHandler (*logging* 中的類🄵), 791
FileHandler (*urllib.request* 中的類🄵), 1440
fileinput
    module, 926
FileInput (*fileinput* 中的類🄵), 927
FileIO (*io* 中的類🄵), 741
filelineno() (於 *fileinput* 模組中), 927
FileLoader (*importlib.abc* 中的類🄵), 2065
filemode() (於 *stat* 模組中), 476
filename (*doctest.DocTest* 的屬性), 1739
filename (*http.cookiejar.FileCookieJar* 的屬性), 1525
filename (*inspect.FrameInfo* 的屬性), 2029
filename (*inspect.Traceback* 的屬性), 2029
filename (*netrc.NetrcParseError* 的屬性), 646
filename (*OSError* 的屬性), 113
filename (*SyntaxError* 的屬性), 114
filename (*traceback.FrameSummary* 的屬性), 2005
filename (*traceback.TracebackException* 的屬性), 2004
filename (*tracemalloc.Frame* 的屬性), 1900
filename (*zipfile.ZipFile* 的屬性), 596
filename (*zipfile.ZipInfo* 的屬性), 599
filename() (於 *fileinput* 模組中), 927
filename2 (*OSError* 的屬性), 113
filename_only (於 *tabnanny* 模組中), 2143
filename_pattern (*tracemalloc.Filter* 的屬性), 1899
filenames（檔案名稱）
    pathname expansion（路徑名稱展開）, 488
    wildcard expansion（萬用字元展開）, 490
fileno() (*bz2.BZ2File* 的方法), 582
fileno() (*http.client.HTTPResponse* 的方法), 1471
fileno() (*io.IOBase* 的方法), 738
fileno() (*multiprocessing.connection.Connection* 的方法), 998
fileno() (*select.devpoll* 的方法), 1239
fileno() (*select.epoll* 的方法), 1240
fileno() (*select.kqueue* 的方法), 1241
fileno() (*selectors.DevpollSelector* 的方法), 1246
fileno() (*selectors.EpollSelector* 的方法), 1246
fileno() (*selectors.KqueueSelector* 的方法), 1246
fileno() (*socketserver.BaseServer* 的方法), 1505
fileno() (*socket.socket* 的方法), 1193
fileno() (於 *fileinput* 模組中), 927
FileNotFoundError, 117
fileobj (*selectors.SelectorKey* 的屬性), 1244
files() (*importlib.abc.TraversableResources* 的方法), 2068
files() (*importlib.resources.abc.TraversableResources* 的方法), 2083
files() (於 *importlib.metadata* 模組中), 2087
files() (於 *importlib.resources* 模組中), 2079
files_double_event() (*tkinter.filedialog.FileDialog* 的方法), 1594
files_select_event() (*tkinter.filedialog.FileDialog* 的方法), 1594
filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）, **2232**
FileType (*argparse* 中的類🄵), 880
FileWrapper (*wsgiref.types* 中的類🄵), 1433
FileWrapper (*wsgiref.util* 中的類🄵), 1427
file（檔案）
    byte-code（位元組碼）, 2145
    configuration（設定）, 626
    copying（🄵🄵）, 492
    debugger（偵錯器）configuration（設定）, 1872
    .ini, 626
    large files（大型檔案）, 2196
    mime.types, 1349
    modes（模式）, 22
    path（路徑）configuration（設定）, 2044
    .pdbrc, 1872
    plist, 647
    temporary（臨時）, 483
fill() (*textwrap.TextWrapper* 的方法), 176
fill() (於 *textwrap* 模組中), 173
fill() (於 *turtle* 模組中), 1645

fillcolor() (於 *turtle* 模組中), 1644
filling() (於 *turtle* 模組中), 1645
fillvalue (*reprlib.Repr* 的屬性), 317
--filter
    tarfile 命令列選項, 616
Filter (*logging* 中的類🄵), 769
filter (*select.kevent* 的屬性), 1242
Filter (*tracemalloc* 中的類🄵), 1899
filter()
    built-in function, 15
filter() (*logging.Filter* 的方法), 769
filter() (*logging.Handler* 的方法), 767
filter() (*logging.Logger* 的方法), 765
filter() (於 *curses* 模組中), 930
filter() (於 *fnmatch* 模組中), 491
filter_command() (*tkinter.filedialog.FileDialog* 的方法), 1594
FILTER_DIR (於 *unittest.mock* 模組中), 1809
filter_traces() (*tracemalloc.Snapshot* 的方法), 1900
FilterError, 604
filterfalse() (於 *fnmatch* 模組中), 491
filterfalse() (於 *itertools* 模組中), 414
filterwarnings() (於 *warnings* 模組中), 1967
Final (於 *typing* 模組中), 1684
final() (於 *typing* 模組中), 1709
finalize (*weakref* 中的類🄵), 299
finalize_dict() (於 *compression.zstd* 模組中), 567
find() (*bytearray* 的方法), 71
find() (*bytes* 的方法), 71
find() (*doctest.DocTestFinder* 的方法), 1740
find() (*mmap.mmap* 的方法), 1258
find() (*str* 的方法), 56
find() (於 *gettext* 模組中), 1562
find() (*xml.etree.ElementTree.Element* 的方法), 1378
find() (*xml.etree.ElementTree.ElementTree* 的方法), 1379
find_class() (*pickle.Unpickler* 的方法), 510
find_class() (pickle 協定), 518
find_library() (於 *ctypes.util* 模組中), 843
find_longest_match() (*difflib.SequenceMatcher* 的方法), 165
find_msvcrt() (於 *ctypes.util* 模組中), 843
find_spec() (*importlib.abc.MetaPathFinder* 的方法), 2062
find_spec() (*importlib.abc.PathEntryFinder* 的方法), 2062
find_spec() (*importlib.machinery.FileFinder* 的方法), 2070
find_spec() (*importlib.machinery.PathFinder* 的類🄵方法), 2069
find_spec() (於 *importlib.util* 模組中), 2074
find_spec() (*zipimport.zipimporter* 的方法), 2052
find_unused_port() (於 *test.support.socket_helper* 模組中), 1846
find_user_password() (*urllib.request.HTTPPasswordMgr* 的方法), 1445
find_user_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 的方法), 1446
findall() (*re.Pattern* 的方法), 152
findall() (於 *re* 模組中), 149
findall() (*xml.etree.ElementTree.Element* 的方法), 1378
findall() (*xml.etree.ElementTree.ElementTree* 的方法), 1379
findCaller() (*logging.Logger* 的方法), 765
finder (尋檢器), **2232**
findfile() (於 *test.support* 模組中), 1839
finditer() (*re.Pattern* 的方法), 152
finditer() (於 *re* 模組中), 150
findlabels() (於 *dis* 模組中), 2155
findlinestarts() (於 *dis* 模組中), 2155
findtext() (*xml.etree.ElementTree.Element* 的方法), 1378
findtext() (*xml.etree.ElementTree.ElementTree* 的方法), 1379
finish() (*socketserver.BaseRequestHandler* 的方法), 1507
finish() (*tkinter.dnd.DndHandler* 的方法), 1598
finish_request() (*socketserver.BaseServer* 的方法), 1506
FIRST_COMPLETED (於 *asyncio* 模組中), 1090
FIRST_COMPLETED (於 *concurrent.futures* 模組中), 1038
FIRST_EXCEPTION (於 *asyncio* 模組中), 1090
FIRST_EXCEPTION (於 *concurrent.futures* 模組中), 1038
firstChild (*xml.dom.Node* 的屬性), 1387
FirstHeaderLineIsContinuationDefect, 1285
firstkey() (*dbm.gnu.gdbm* 的方法), 529
--first-weekday
    calendar 命令列選項, 263
firstweekday (*calendar.Calendar* 的屬性), 256
firstweekday() (於 *calendar* 模組中), 259
fix_missing_locations() (於 *ast* 模組中), 2126
fix_sentence_endings (*textwrap.TextWrapper* 的屬性), 175
Flag (*enum* 中的類🄵), 327
flag_bits (*zipfile.ZipInfo* 的屬性), 600
FlagBoundary (*enum* 中的類🄵), 330
flags (*decimal.Context* 的屬性), 366
flags (*re.Pattern* 的屬性), 153
flags (*select.kevent* 的屬性), 1242
flags (於 *sys* 模組中), 1925
flash() (於 *curses* 模組中), 930
flatten() (*email.generator.BytesGenerator* 的方法), 1276
flatten() (*email.generator.Generator* 的方法), 1277
flattening (攤平)
    objects (物件), 505
float
    built-in function (🄵建函式), 38
--float
    random 命令列選項, 393

float (🄵建類🄵), 16
float_info (於 sys 模組中), 1927
float_repr_style (於 sys 模組中), 1929
FloatingPointError, 111
floating-point (浮點數)
    literals (字面值), 38
    object (物件), 38
FloatOperation (decimal 中的類🄵), 373
flock() (於 fcntl 模組中), 2205
floor division (向下取整除法), **2233**
floor() (於 math 模組中), 342
floor() (於 math 模組), 39
FloorDiv (ast 中的類🄵), 2102
floordiv() (於 operator 模組中), 436
flush() (bz2.BZ2Compressor 的方法), 583
flush() (compression.zstd.ZstdCompressor 的方法), 566
flush() (io.BufferedWriter 的方法), 742
flush() (io.IOBase 的方法), 738
flush() (logging.Handler 的方法), 767
flush() (logging.handlers.BufferingHandler 的方法), 800
flush() (logging.handlers.MemoryHandler 的方法), 801
flush() (logging.StreamHandler 的方法), 791
flush() (lzma.LZMACompressor 的方法), 588
flush() (mailbox.Mailbox 的方法), 1332
flush() (mailbox.Maildir 的方法), 1334
flush() (mailbox.MH 的方法), 1337
flush() (mmap.mmap 的方法), 1258
flush() (xml.etree.ElementTree.XMLParser 的方法), 1382
flush() (xml.etree.ElementTree.XMLPullParser 的方法), 1383
flush() (zlib.Compress 的方法), 576
flush() (zlib.Decompress 的方法), 577
FLUSH_BLOCK (compression.zstd.ZstdCompressor 的屬性), 566
FLUSH_FRAME (compression.zstd.ZstdCompressor 的屬性), 566
flush_headers() (http.server.BaseHTTPRequestHandler 的方法), 1514
flush_std_streams() (於 test.support 模組中), 1841
flushinp() (於 curses 模組中), 930
FlushKey() (於 winreg 模組中), 2182
fma() (decimal.Context 的方法), 368
fma() (decimal.Decimal 的方法), 360
fma() (於 math 模組中), 342
fmean() (於 statistics 模組中), 395
fmod() (於 math 模組中), 342
FMT_BINARY (於 plistlib 模組中), 649
FMT_XML (於 plistlib 模組中), 648
fnmatch
    module, 490
fnmatch() (於 fnmatch 模組中), 491
fnmatchcase() (於 fnmatch 模組中), 491
focus() (tkinter.ttk.Treeview 的方法), 1610
fold (datetime.datetime 的屬性), 228
fold (datetime.time 的屬性), 236
fold() (email.headerregistry.BaseHeader 的方法), 1287
fold() (email.policy.Compat32 的方法), 1284
fold() (email.policy.EmailPolicy 的方法), 1283
fold() (email.policy.Policy 的方法), 1281
fold_binary() (email.policy.Compat32 的方法), 1284
fold_binary() (email.policy.EmailPolicy 的方法), 1283
fold_binary() (email.policy.Policy 的方法), 1281
Font (tkinter.font 中的類🄵), 1591
For (ast 中的類🄵), 2110
FOR_ITER (opcode), 2166
forget() (tkinter.ttk.Notebook 的方法), 1605
forget() (於 test.support.import_helper 模組中), 1851
fork() (於 os 模組中), 719
fork() (於 pty 模組中), 2201
ForkingMixIn (socketserver 中的類🄵), 1504
ForkingTCPServer (socketserver 中的類🄵), 1504
ForkingUDPServer (socketserver 中的類🄵), 1504
ForkingUnixDatagramServer (socketserver 中的類🄵), 1504
ForkingUnixStreamServer (socketserver 中的類🄵), 1504
forkpty() (於 os 模組中), 719
Format (annotationlib 中的類🄵), 2036
FORMAT (inspect.BufferFlags 的屬性), 2034
format (memoryview 的屬性), 88
format (multiprocessing.shared_memory.ShareableList 的屬性), 1030
format (struct.Struct 的屬性), 193
format()
    built-in function, 16
format() (inspect.Signature 的方法), 2024
format() (logging.BufferingFormatter 的方法), 769
format() (logging.Formatter 的方法), 768
format() (logging.Handler 的方法), 767
format() (pprint.PrettyPrinter 的方法), 313
format() (str 的方法), 56
format() (string.Formatter 的方法), 124
format() (traceback.StackSummary 的方法), 2005
format() (traceback.TracebackException 的方法), 2004
format() (tracemalloc.Traceback 的方法), 1902
format_call_graph() (於 asyncio 模組中), 1119
format_datetime() (於 email.utils 模組中), 1318
format_exc() (於 traceback 模組中), 2002
format_exception() (於 traceback 模組中), 2002
format_exception_only() (traceback.TracebackException 的方法), 2004
format_exception_only() (於 traceback 模組中), 2002
format_field() (string.Formatter 的方法), 125
format_frame_summary() (traceback.StackSummary 的方法), 2005

format_help() (*argparse.ArgumentParser* 的方法), 883
format_list() (於 *traceback* 模組中), 2002
format_map() (*str* 的方法), 56
FORMAT_SIMPLE (*opcode*), 2170
format_spec (*string.templatelib.Interpolation* 的屬性), 138
format_stack() (於 *traceback* 模組中), 2002
format_stack_entry() (*bdb.Bdb* 的方法), 1864
format_string() (於 *locale* 模組中), 1574
format_tb() (於 *traceback* 模組中), 2002
format_usage() (*argparse.Action* 的方法), 873
format_usage() (*argparse.ArgumentParser* 的方法), 883
FORMAT_WITH_SPEC (*opcode*), 2170
formataddr() (於 *email.utils* 模組中), 1316
formatargvalues() (於 *inspect* 模組中), 2027
formatdate() (於 *email.utils* 模組中), 1317
formatday() (*calendar.TextCalendar* 的方法), 257
FormatError, 1347
FormatError() (於 *ctypes* 模組中), 844
formatException() (*logging.Formatter* 的方法), 769
formatFooter() (*logging.BufferingFormatter* 的方法), 769
formatHeader() (*logging.BufferingFormatter* 的方法), 769
formatmonth() (*calendar.HTMLCalendar* 的方法), 258
formatmonth() (*calendar.TextCalendar* 的方法), 257
formatmonthname() (*calendar.HTMLCalendar* 的方法), 258
formatmonthname() (*calendar.TextCalendar* 的方法), 257
formatStack() (*logging.Formatter* 的方法), 769
formatted string literals (格式化字串常數), 63
FormattedValue (*ast* 中的類🄵), 2098
Formatter (*logging* 中的類🄵), 768
Formatter (*string* 中的類🄵), 124
formatTime() (*logging.Formatter* 的方法), 768
formatting (格式化)
    bytearray (%), 80
    bytes (%), 80
formatting (格式化)、字串 (%), 64
formatwarning() (於 *warnings* 模組中), 1967
formatweek() (*calendar.TextCalendar* 的方法), 257
formatweekday() (*calendar.TextCalendar* 的方法), 257
formatweekheader() (*calendar.TextCalendar* 的方法), 257
formatyear() (*calendar.HTMLCalendar* 的方法), 258
formatyear() (*calendar.TextCalendar* 的方法), 257
formatyearpage() (*calendar.HTMLCalendar* 的方法), 258
Fortran contiguous (Fortran 連續的), 2230
forward() (於 *turtle* 模組中), 1635
ForwardRef (*annotationlib* 中的類🄵), 2037
FORWARDREF (*annotationlib.Format* 的屬性), 2037
ForwardRef (*typing* 中的類🄵), 1713
fp (*urllib.error.HTTPError* 的屬性), 1461
fpathconf() (於 *os* 模組中), 679
Fraction (*fractions* 中的類🄵), 381
fractions
    module, 381
frame (*inspect.FrameInfo* 的屬性), 2029
frame (*tkinter.scrolledtext.ScrolledText* 的屬性), 1598
Frame (*tracemalloc* 中的類🄵), 1900
FrameInfo (*compression.zstd* 中的類🄵), 573
FrameInfo (*inspect* 中的類🄵), 2029
FrameSummary (*traceback* 中的類🄵), 2005
FrameType (於 *types* 模組中), 307
free threading (自由執行緒), **2233**
free variable (自由變數), **2233**
free_tool_id() (於 *sys.monitoring* 模組中), 1947
freedesktop_os_release() (於 *platform* 模組中), 807
freeze() (於 *gc* 模組中), 2014
freeze_support() (於 *multiprocessing* 模組中), 996
frexp() (於 *math* 模組中), 343
FRIDAY (於 *calendar* 模組中), 260
from_address() (*ctypes._CData* 的方法), 846
from_buffer() (*ctypes._CData* 的方法), 845
from_buffer_copy() (*ctypes._CData* 的方法), 845
from_bytes() (*int* 的類🄵方法), 41
from_callable() (*inspect.Signature* 的類🄵方法), 2024
from_decimal() (*fractions.Fraction* 的類🄵方法), 383
from_exception() (*traceback.TracebackException* 的類🄵方法), 2004
from_file() (*zipfile.ZipInfo* 的類🄵方法), 599
from_file() (*zoneinfo.ZoneInfo* 的類🄵方法), 253
from_float() (*decimal.Decimal* 的類🄵方法), 360
from_float() (*fractions.Fraction* 的類🄵方法), 382
from_iterable() (*itertools.chain* 的類🄵方法), 412
from_list() (*traceback.StackSummary* 的類🄵方法), 2005
from_number() (*complex* 的類🄵方法), 43
from_number() (*decimal.Decimal* 的類🄵方法), 360
from_number() (*float* 的類🄵方法), 42
from_number() (*fractions.Fraction* 的類🄵方法), 383
from_param() (*ctypes._CData* 的方法), 846
from_samples() (*statistics.NormalDist* 的類🄵方法), 404
from_traceback() (*dis.Bytecode* 的類🄵方法), 2153
from_uri() (*pathlib.Path* 的類🄵方法), 455
frombuf() (*tarfile.TarInfo* 的類🄵方法), 610
frombytes() (*array.array* 的方法), 295
fromfd() (*select.epoll* 的方法), 1240
fromfd() (*select.kqueue* 的方法), 1242
fromfd() (於 *socket* 模組中), 1185
fromfile() (*array.array* 的方法), 295
fromhex() (*bytearray* 的類🄵方法), 68
fromhex() (*bytes* 的類🄵方法), 67
fromhex() (*float* 的類🄵方法), 43

fromisocalendar() (*datetime.date* 的類🄵方法), 220
fromisocalendar() (*datetime.datetime* 的類🄵方法), 227
fromisoformat() (*datetime.date* 的類🄵方法), 220
fromisoformat() (*datetime.datetime* 的類🄵方法), 227
fromisoformat() (*datetime.time* 的類🄵方法), 237
fromkeys() (*collections.Counter* 的方法), 267
fromkeys() (*dict* 的類🄵方法), 93
fromlist() (*array.array* 的方法), 295
fromordinal() (*datetime.date* 的類🄵方法), 220
fromordinal() (*datetime.datetime* 的類🄵方法), 226
fromshare() (於 *socket* 模組中), 1186
fromstring() (於 *xml.etree.ElementTree* 模組中), 1373
fromstringlist() (於 *xml.etree.ElementTree* 模組中), 1373
fromtarfile() (*tarfile.TarInfo* 的類🄵方法), 610
fromtimestamp() (*datetime.date* 的類🄵方法), 219
fromtimestamp() (*datetime.datetime* 的類🄵方法), 225
fromunicode() (*array.array* 的方法), 295
fromutc() (*datetime.timezone* 的方法), 246
fromutc() (*datetime.tzinfo* 的方法), 241
FrozenImporter (*importlib.machinery* 中的類🄵), 2069
FrozenInstanceError, 1976
FrozenSet (*typing* 中的類🄵), 1715
frozenset (🄵建類🄵), 89
FS (於 *curses.ascii* 模組中), 956
fs_is_case_insensitive() (於 *test.support.os_helper* 模組中), 1850
FS_NONASCII (於 *test.support.os_helper* 模組中), 1849
fsdecode() (於 *os* 模組中), 670
fsencode() (於 *os* 模組中), 670
fspath() (於 *os* 模組中), 670
fstat() (於 *os* 模組中), 679
fstatvfs() (於 *os* 模組中), 679
FSTRING_END (於 *token* 模組中), 2135
FSTRING_MIDDLE (於 *token* 模組中), 2135
FSTRING_START (於 *token* 模組中), 2134
fstring (f 字串), 63
f-string (f 字串), 63
fsum() (於 *math* 模組中), 346
fsync() (於 *os* 模組中), 679
FTP, 1451
    ftplib (標准模組), 1474
    protocol (協定), 1451, 1474
FTP (*ftplib* 中的類🄵), 1474
ftp_open() (*urllib.request.FTPHandler* 的方法), 1447
FTP_TLS (*ftplib* 中的類🄵), 1478
FTPHandler (*urllib.request* 中的類🄵), 1440
ftplib
    module, 1474
ftruncate() (於 *os* 模組中), 679
Full, 1065
FULL (*inspect.BufferFlags* 的屬性), 2034
full() (*asyncio.Queue* 的方法), 1116
full() (*multiprocessing.Queue* 的方法), 994
full() (*queue.Queue* 的方法), 1065
full_match() (*pathlib.PurePath* 的方法), 451
FULL_RO (*inspect.BufferFlags* 的屬性), 2034
full_url (*urllib.request.Request* 的屬性), 1441
fullmatch() (*re.Pattern* 的方法), 152
fullmatch() (於 *re* 模組中), 149
fully_trusted_filter() (於 *tarfile* 模組中), 613
func (*functools.partial* 的屬性), 435
funcname (*bdb.Breakpoint* 的屬性), 1860
function (*inspect.FrameInfo* 的屬性), 2029
function (*inspect.Traceback* 的屬性), 2029
Function (*pyclbr* 中的類🄵), 2144
Function (*symtable* 中的類🄵), 2131
FUNCTION (*symtable.SymbolTableType* 的屬性), 2130
function annotation (函式🄵釋), **2233**
FunctionDef (*ast* 中的類🄵), 2121
FunctionTestCase (*unittest* 中的類🄵), 1766
FunctionType (*ast* 中的類🄵), 2098
FunctionType (於 *types* 模組中), 305
function (函式), **2233**
functools
    module, 425
funny_files (*filecmp.dircmp* 的屬性), 482
Future (*asyncio* 中的類🄵), 1144
Future (*concurrent.futures* 中的類🄵), 1037
future_add_to_awaited_by() (於 *asyncio* 模組中), 1120
future_discard_from_awaited_by() (於 *asyncio* 模組中), 1120
FutureWarning, 118
fwalk() (於 *os* 模組中), 709

G

-g
    trace 命令列選項, 1891
gaierror, 1177
gamma() (於 *math* 模組中), 348
gammavariate() (於 *random* 模組中), 388
garbage (於 *gc* 模組中), 2015
garbage collection (垃圾回收), **2233**
gather() (*curses.textpad.Textbox* 的方法), 954
gather() (於 *asyncio* 模組中), 1085
gauss() (於 *random* 模組中), 388
gc
    module, 2012
gc_collect() (於 *test.support* 模組中), 1840
gcd() (於 *math* 模組中), 342
ge() (於 *operator* 模組中), 435
generate_tokens() (於 *tokenize* 模組中), 2140
Generator (*collections.abc* 中的類🄵), 283
Generator (*email.generator* 中的類🄵), 1276
Generator (*typing* 中的類🄵), 1719
generator expression (🄵生器運算式), 2233, **2234**
generator iterator (🄵生器🄵代器), **2233**
generator_stop (於 *__future__* 模組中), 2011

GeneratorExit, 111
GeneratorExp (*ast* 中的類🄵), 2105
generators (於 *__future__* 模組中), 2011
GeneratorType (於 *types* 模組中), 305
generator（🄵生器）, **2233**
Generic (*typing* 中的類🄵), 1690
generic function（泛型函式）, **2234**
generic type（泛型型🄵）, **2234**
generic_visit() (*ast.NodeVisitor* 的方法), 2126
GenericAlias (*types* 中的類🄵), 306
GenericAlias（泛型🄵名）
    object（物件）, 97
Generic（泛型）
    Alias（🄵名）, 97
genops() (於 *pickletools* 模組中), 2175
geometric_mean() (於 *statistics* 模組中), 396
get() (*asyncio.Queue* 的方法), 1116
get() (*configparser.ConfigParser* 的方法), 641
get() (*contextvars.Context* 的方法), 1070
get() (*contextvars.ContextVar* 的方法), 1068
get() (*dict* 的方法), 93
get() (*email.message.EmailMessage* 的方法), 1266
get() (*email.message.Message* 的方法), 1304
get() (*mailbox.Mailbox* 的方法), 1331
get() (*multiprocessing.pool.AsyncResult* 的方法), 1012
get() (*multiprocessing.Queue* 的方法), 994
get() (*multiprocessing.SimpleQueue* 的方法), 995
get() (*queue.Queue* 的方法), 1065
get() (*queue.SimpleQueue* 的方法), 1067
get() (*tkinter.ttk.Combobox* 的方法), 1603
get() (*tkinter.ttk.Spinbox* 的方法), 1603
get() (*types.MappingProxyType* 的方法), 308
get() (於 *webbrowser* 模組中), 1424
get() (*xml.etree.ElementTree.Element* 的方法), 1377
GET_AITER (*opcode*), 2159
get_all() (*email.message.EmailMessage* 的方法), 1267
get_all() (*email.message.Message* 的方法), 1304
get_all() (*wsgiref.headers.Headers* 的方法), 1428
get_all_breaks() (*bdb.Bdb* 的方法), 1863
get_all_start_methods() (於 *multiprocessing* 模組中), 996
GET_ANEXT (*opcode*), 2160
get_annotate_from_class_namespace() (於 *annotationlib* 模組中), 2039
get_annotations() (於 *annotationlib* 模組中), 2039
get_annotations() (於 *inspect* 模組中), 2028
get_app() (*wsgiref.simple_server.WSGIServer* 的方法), 1429
get_archive_formats() (於 *shutil* 模組中), 499
get_args() (於 *typing* 模組中), 1712
get_asyncgen_hooks() (於 *sys* 模組中), 1932
get_attribute() (於 *test.support* 模組中), 1843
GET_AWAITABLE (*opcode*), 2159
get_begidx() (於 *readline* 模組中), 183
get_blocking() (於 *os* 模組中), 680
get_body() (*email.message.EmailMessage* 的方法), 1270
get_body_encoding() (*email.charset.Charset* 的方法), 1314
get_boundary() (*email.message.EmailMessage* 的方法), 1268
get_boundary() (*email.message.Message* 的方法), 1306
get_bpbynumber() (*bdb.Bdb* 的方法), 1863
get_break() (*bdb.Bdb* 的方法), 1863
get_breaks() (*bdb.Bdb* 的方法), 1863
get_buffer() (*asyncio.BufferedProtocol* 的方法), 1153
get_bytes() (*mailbox.Mailbox* 的方法), 1331
get_bytes() (*mailbox.mbox* 的方法), 1335
get_bytes() (*mailbox.MMDF* 的方法), 1338
get_ca_certs() (*ssl.SSLContext* 的方法), 1220
get_cache_token() (於 *abc* 模組中), 1998
get_channel_binding() (*ssl.SSLSocket* 的方法), 1217
get_charset() (*email.message.Message* 的方法), 1303
get_charsets() (*email.message.EmailMessage* 的方法), 1268
get_charsets() (*email.message.Message* 的方法), 1307
get_children() (*symtable.SymbolTable* 的方法), 2131
get_children() (*tkinter.ttk.Treeview* 的方法), 1609
get_ciphers() (*ssl.SSLContext* 的方法), 1220
get_clock_info() (於 *time* 模組中), 750
get_close_matches() (於 *difflib* 模組中), 163
get_code() (*importlib.abc.InspectLoader* 的方法), 2064
get_code() (*importlib.abc.SourceLoader* 的方法), 2066
get_code() (*importlib.machinery.ExtensionFileLoader* 的方法), 2071
get_code() (*importlib.machinery.SourcelessFileLoader* 的方法), 2071
get_code() (*zipimport.zipimporter* 的方法), 2052
get_completer() (於 *readline* 模組中), 183
get_completer_delims() (於 *readline* 模組中), 183
get_completion_type() (於 *readline* 模組中), 183
get_config_h_filename() (於 *sysconfig* 模組中), 1956
get_config_var() (於 *sysconfig* 模組中), 1951
get_config_vars() (於 *sysconfig* 模組中), 1951
get_content() (*email.contentmanager.ContentManager* 的方法), 1292
get_content() (*email.message.EmailMessage* 的方法), 1270
get_content() (於 *email.contentmanager* 模組中), 1293
get_content_charset() (*email.message.EmailMessage* 的方法), 1268
get_content_charset() (*email.message.Message* 的方法), 1307
get_content_disposition()

(*email.message.EmailMessage* 的方法), 1269
get_content_disposition() (*email.message.Message* 的方法), 1307
get_content_maintype() (*email.message.EmailMessage* 的方法), 1267
get_content_maintype() (*email.message.Message* 的方法), 1305
get_content_subtype() (*email.message.EmailMessage* 的方法), 1267
get_content_subtype() (*email.message.Message* 的方法), 1305
get_content_type() (*email.message.EmailMessage* 的方法), 1267
get_content_type() (*email.message.Message* 的方法), 1305
get_context() (*asyncio.Handle* 的方法), 1138
get_context() (*asyncio.Task* 的方法), 1096
get_context() (於 *multiprocessing* 模組中), 996
get_coro() (*asyncio.Task* 的方法), 1095
get_coroutine_origin_tracking_depth() (於 *sys* 模組中), 1932
get_count() (於 *gc* 模組中), 2013
get_current() (於 *concurrent.interpreters* 模組中), 1042
get_current_history_length() (於 *readline* 模組中), 182
get_data() (*importlib.abc.FileLoader* 的方法), 2065
get_data() (*importlib.abc.ResourceLoader* 的方法), 2063
get_data() (於 *pkgutil* 模組中), 2055
get_data() (*zipimport.zipimporter* 的方法), 2052
get_date() (*mailbox.MaildirMessage* 的方法), 1340
get_debug() (*asyncio.loop* 的方法), 1137
get_debug() (於 *gc* 模組中), 2013
get_default() (*argparse.ArgumentParser* 的方法), 883
get_default_backend() (於 *pdb* 模組中), 1871
get_default_scheme() (於 *sysconfig* 模組中), 1955
get_default_type() (*email.message.EmailMessage* 的方法), 1267
get_default_type() (*email.message.Message* 的方法), 1305
get_default_verify_paths() (於 *ssl* 模組中), 1207
get_dialect() (於 *csv* 模組中), 620
get_disassembly_as_string() (*test.support.bytecode_helper.BytecodeTestCase* 的方法), 1848
get_docstring() (於 *ast* 模組中), 2126
get_doctest() (*doctest.DocTestParser* 的方法), 1740
get_endidx() (於 *readline* 模組中), 183
get_environ() (*wsgiref.simple_server.WSGIRequestHandler* 的方法), 1429
get_errno() (於 *ctypes* 模組中), 844
get_escdelay() (於 *curses* 模組中), 934
get_event_loop() (*asyncio.AbstractEventLoopPolicy* 的方法), 1160
get_event_loop() (於 *asyncio* 模組中), 1121
get_event_loop_policy() (於 *asyncio* 模組中), 1160
get_events() (於 *sys.monitoring* 模組中), 1950
get_examples() (*doctest.DocTestParser* 的方法), 1740
get_exception_handler() (*asyncio.loop* 的方法), 1136
get_exec_path() (於 *os* 模組中), 671
get_extra_info() (*asyncio.BaseTransport* 的方法), 1148
get_extra_info() (*asyncio.StreamWriter* 的方法), 1102
get_field() (*string.Formatter* 的方法), 124
get_file() (*mailbox.Babyl* 的方法), 1338
get_file() (*mailbox.Mailbox* 的方法), 1331
get_file() (*mailbox.Maildir* 的方法), 1335
get_file() (*mailbox.mbox* 的方法), 1335
get_file() (*mailbox.MH* 的方法), 1337
get_file() (*mailbox.MMDF* 的方法), 1338
get_file_breaks() (*bdb.Bdb* 的方法), 1863
get_filename() (*email.message.EmailMessage* 的方法), 1268
get_filename() (*email.message.Message* 的方法), 1306
get_filename() (*importlib.abc.ExecutionLoader* 的方法), 2065
get_filename() (*importlib.abc.FileLoader* 的方法), 2065
get_filename() (*importlib.machinery.ExtensionFileLoader* 的方法), 2072
get_filename() (*zipimport.zipimporter* 的方法), 2052
get_filter() (*tkinter.filedialog.FileDialog* 的方法), 1594
get_flags() (*mailbox.Maildir* 的方法), 1333
get_flags() (*mailbox.MaildirMessage* 的方法), 1340
get_flags() (*mailbox.mboxMessage* 的方法), 1341
get_flags() (*mailbox.MMDFMessage* 的方法), 1345
get_folder() (*mailbox.Maildir* 的方法), 1333
get_folder() (*mailbox.MH* 的方法), 1336
get_frame_info() (於 *compression.zstd* 模組中), 573
get_frees() (*symtable.Function* 的方法), 2131
get_freeze_count() (於 *gc* 模組中), 2015
get_from() (*mailbox.mboxMessage* 的方法), 1341
get_from() (*mailbox.MMDFMessage* 的方法), 1345
get_full_url() (*urllib.request.Request* 的方法), 1442
get_globals() (*symtable.Function* 的方法), 2131
get_grouped_opcodes() (*difflib.SequenceMatcher* 的方法), 167
get_handle_inheritable() (於 *os* 模組中), 690
get_header() (*urllib.request.Request* 的方法), 1442

get_history_item() (於 readline 模組中), 182
get_history_length() (於 readline 模組中), 181
get_id() (symtable.SymbolTable 的方法), 2130
get_ident() (於 _thread 模組中), 1072
get_ident() (於 threading 模組中), 967
get_identifiers() (string.Template 的方法), 133
get_identifiers() (symtable.SymbolTable 的方法), 2131
get_importer() (於 pkgutil 模組中), 2053
get_info() (mailbox.Maildir 的方法), 1334
get_info() (mailbox.MaildirMessage 的方法), 1340
get_inheritable() (socket.socket 的方法), 1193
get_inheritable() (於 os 模組中), 690
get_instructions() (於 dis 模組中), 2155
get_int_max_str_digits() (於 sys 模組中), 1930
get_interpreter() (於 zipapp 模組中), 1916
GET_ITER (opcode), 2158
get_key() (selectors.BaseSelector 的方法), 1246
get_labels() (mailbox.Babyl 的方法), 1338
get_labels() (mailbox.BabylMessage 的方法), 1344
get_last_error() (於 ctypes 模組中), 844
GET_LEN (opcode), 2162
get_line_buffer() (於 readline 模組中), 181
get_lineno() (symtable.SymbolTable 的方法), 2131
get_local_events() (於 sys.monitoring 模組中), 1950
get_locals() (symtable.Function 的方法), 2131
get_logger() (於 multiprocessing 模組中), 1016
get_loop() (asyncio.Future 的方法), 1146
get_loop() (asyncio.Runner 的方法), 1078
get_loop() (asyncio.Server 的方法), 1140
get_main() (於 concurrent.interpreters 模組中), 1042
get_makefile_filename() (於 sysconfig 模組中), 1956
get_map() (selectors.BaseSelector 的方法), 1246
get_matching_blocks() (difflib.SequenceMatcher 的方法), 166
get_message() (mailbox.Mailbox 的方法), 1331
get_method() (urllib.request.Request 的方法), 1441
get_methods() (symtable.Class 的方法), 2131
get_mixed_type_key() (於 ipaddress 模組中), 1556
get_name() (asyncio.Task 的方法), 1096
get_name() (symtable.Symbol 的方法), 2132
get_name() (symtable.SymbolTable 的方法), 2130
get_namespace() (symtable.Symbol 的方法), 2133
get_namespaces() (symtable.Symbol 的方法), 2133
get_native_id() (於 _thread 模組中), 1072
get_native_id() (於 threading 模組中), 967
get_nonlocals() (symtable.Function 的方法), 2131
get_nonstandard_attr() (http.cookiejar.Cookie 的方法), 1529
get_nowait() (asyncio.Queue 的方法), 1116
get_nowait() (multiprocessing.Queue 的方法), 994
get_nowait() (queue.Queue 的方法), 1066
get_nowait() (queue.SimpleQueue 的方法), 1067
get_object_traceback() (於 tracemalloc 模組中), 1897
get_objects() (於 gc 模組中), 2013
get_opcodes() (difflib.SequenceMatcher 的方法), 166
get_option() (optparse.OptionParser 的方法), 916
get_option_group() (optparse.OptionParser 的方法), 907
get_origin() (於 typing 模組中), 1711
get_original_bases() (於 types 模組中), 304
get_original_stdout() (於 test.support 模組中), 1840
get_osfhandle() (於 msvcrt 模組中), 2178
get_output_charset() (email.charset.Charset 的方法), 1314
get_overloads() (於 typing 模組中), 1709
get_pagesize() (於 test.support 模組中), 1839
get_param() (email.message.Message 的方法), 1305
get_parameters() (symtable.Function 的方法), 2131
get_params() (email.message.Message 的方法), 1305
get_path() (於 sysconfig 模組中), 1955
get_path_names() (於 sysconfig 模組中), 1955
get_paths() (於 sysconfig 模組中), 1955
get_payload() (email.message.Message 的方法), 1302
get_pid() (asyncio.SubprocessTransport 的方法), 1151
get_pipe_transport() (asyncio.SubprocessTransport 的方法), 1151
get_platform() (於 sysconfig 模組中), 1956
get_poly() (於 turtle 模組中), 1650
get_preferred_scheme() (於 sysconfig 模組中), 1955
get_protocol() (asyncio.BaseTransport 的方法), 1149
get_protocol_members() (於 typing 模組中), 1712
get_proxy_response_headers() (http.client.HTTPConnection 的方法), 1470
get_python_version() (於 sysconfig 模組中), 1956
get_ready() (graphlib.TopologicalSorter 的方法), 336
get_referents() (於 gc 模組中), 2014
get_referrers() (於 gc 模組中), 2013
get_request() (socketserver.BaseServer 的方法), 1506
get_returncode() (asyncio.SubprocessTransport 的方法), 1151
get_running_loop() (於 asyncio 模組中), 1121
get_scheme() (wsgiref.handlers.BaseHandler 的方法), 1432
get_scheme_names() (於 sysconfig 模組中), 1955
get_selection() (tkinter.filedialog.FileDialog 的方法), 1594
get_sequences() (mailbox.MH 的方法), 1336
get_sequences() (mailbox.MHMessage 的方法), 1343
get_server() (multiprocessing.managers.BaseManager 的方法), 1005
get_server_certificate() (於 ssl 模組中), 1206
get_shapepoly() (於 turtle 模組中), 1649
get_source() (importlib.abc.InspectLoader 的方法),

2064
get_source() (*importlib.abc.SourceLoader* 的方法), 2066
get_source() (*importlib.machinery.ExtensionFileLoader* 的方法), 2071
get_source() (*importlib.machinery.SourcelessFileLoader* 的方法), 2071
get_source() (*zipimport.zipimporter* 的方法), 2052
get_source_segment() (於 *ast* 模組中), 2126
get_stack() (*asyncio.Task* 的方法), 1095
get_stack() (*bdb.Bdb* 的方法), 1863
get_start_method() (於 *multiprocessing* 模組中), 996
get_starttag_text() (*html.parser.HTMLParser* 的方法), 1361
get_stats() (於 *gc* 模組中), 2013
get_stats_profile() (*pstats.Stats* 的方法), 1884
get_stderr() (*wsgiref.handlers.BaseHandler* 的方法), 1431
get_stderr() (*wsgiref.simple_server.WSGIRequestHandler* 的方法), 1429
get_stdin() (*wsgiref.handlers.BaseHandler* 的方法), 1431
get_string() (*mailbox.Mailbox* 的方法), 1331
get_string() (*mailbox.mbox* 的方法), 1335
get_subdir() (*mailbox.MaildirMessage* 的方法), 1340
get_symbols() (*symtable.SymbolTable* 的方法), 2131
get_tabsize() (於 *curses* 模組中), 934
get_task_factory() (*asyncio.loop* 的方法), 1125
get_terminal_size() (於 *os* 模組中), 689
get_terminal_size() (於 *shutil* 模組中), 502
get_threshold() (於 *gc* 模組中), 2013
get_token() (*shlex.shlex* 的方法), 2193
get_tool() (於 *sys.monitoring* 模組中), 1947
get_traceback_limit() (於 *tracemalloc* 模組中), 1897
get_traced_memory() (於 *tracemalloc* 模組中), 1897
get_tracemalloc_memory() (於 *tracemalloc* 模組中), 1898
get_type() (*symtable.SymbolTable* 的方法), 2130
get_type_hints() (於 *typing* 模組中), 1711
get_unixfrom() (*email.message.EmailMessage* 的方法), 1265
get_unixfrom() (*email.message.Message* 的方法), 1302
get_unpack_formats() (於 *shutil* 模組中), 501
get_unverified_chain() (*ssl.SSLSocket* 的方法), 1216
get_usage() (*optparse.OptionParser* 的方法), 918
get_value() (*string.Formatter* 的方法), 124
get_verified_chain() (*ssl.SSLSocket* 的方法), 1216
get_version() (*optparse.OptionParser* 的方法), 908
get_visible() (*mailbox.BabylMessage* 的方法), 1344
get_wch() (*curses.window* 的方法), 938
get_write_buffer_limits() (*asyncio.WriteTransport* 的方法), 1150
get_write_buffer_size() (*asyncio.WriteTransport* 的方法), 1149
GET_YIELD_FROM_ITER (*opcode*), 2158
getacl() (*imaplib.IMAP4* 的方法), 1485
getaddresses() (於 *email.utils* 模組中), 1317
getaddrinfo() (*asyncio.loop* 的方法), 1133
getaddrinfo() (於 *socket* 模組中), 1186
getallocatedblocks() (於 *sys* 模組中), 1929
getandroidapilevel() (於 *sys* 模組中), 1929
getannotation() (*imaplib.IMAP4* 的方法), 1485
getargvalues() (於 *inspect* 模組中), 2027
getasyncgenlocals() (於 *inspect* 模組中), 2033
getasyncgenstate() (於 *inspect* 模組中), 2032
getatime() (於 *os.path* 模組中), 470
getattr()
    built-in function, 17
getattr_static() (於 *inspect* 模組中), 2031
getAttribute() (*xml.dom.Element* 的方法), 1390
getAttributeNode() (*xml.dom.Element* 的方法), 1390
getAttributeNodeNS() (*xml.dom.Element* 的方法), 1390
getAttributeNS() (*xml.dom.Element* 的方法), 1390
GetBase() (*xml.parsers.expat.xmlparser* 的方法), 1413
getbegyx() (*curses.window* 的方法), 938
getbkgd() (*curses.window* 的方法), 938
getblocking() (*socket.socket* 的方法), 1193
getboolean() (*configparser.ConfigParser* 的方法), 641
getbuffer() (*io.BytesIO* 的方法), 741
getByteStream() (*xml.sax.xmlreader.InputSource* 的方法), 1410
getcallargs() (於 *inspect* 模組中), 2028
getcanvas() (於 *turtle* 模組中), 1657
getch() (*curses.window* 的方法), 938
getch() (於 *msvcrt* 模組中), 2178
getCharacterStream() (*xml.sax.xmlreader.InputSource* 的方法), 1410
getche() (於 *msvcrt* 模組中), 2178
getChild() (*logging.Logger* 的方法), 763
getChildren() (*logging.Logger* 的方法), 763
getclasstree() (於 *inspect* 模組中), 2027
getclosurevars() (於 *inspect* 模組中), 2028
getcode() (*http.client.HTTPResponse* 的方法), 1472
getcode() (*urllib.response.addinfourl* 的方法), 1452
getColumnNumber() (*xml.sax.xmlreader.Locator* 的方法), 1410
getcomments() (於 *inspect* 模組中), 2021
getcompname() (*wave.Wave_read* 的方法), 1558
getcomptype() (*wave.Wave_read* 的方法), 1558
getconfig() (*sqlite3.Connection* 的方法), 546
getContentHandler() (*xml.sax.xmlreader.XMLReader* 的方法), 1408
getcontext() (於 *decimal* 模組中), 365

getcoroutinelocals() (於 inspect 模組中), 2033
getcoroutinestate() (於 inspect 模組中), 2032
getctime() (於 os.path 模組中), 470
getcwd() (於 os 模組中), 693
getcwdb() (於 os 模組中), 693
getdecoder() (於 codecs 模組中), 195
getdefaultencoding() (於 sys 模組中), 1929
getdefaultlocale() (於 locale 模組中), 1573
getdefaulttimeout() (於 socket 模組中), 1190
getdlopenflags() (於 sys 模組中), 1929
getdoc() (於 inspect 模組中), 2021
getDOMImplementation() (於 xml.dom 模組中), 1385
getDTDHandler() (xml.sax.xmlreader.XMLReader 的方法), 1409
getEffectiveLevel() (logging.Logger 的方法), 762
getegid() (於 os 模組中), 671
getElementsByTagName() (xml.dom.Document 的方法), 1389
getElementsByTagName() (xml.dom.Element 的方法), 1390
getElementsByTagNameNS() (xml.dom.Document 的方法), 1389
getElementsByTagNameNS() (xml.dom.Element 的方法), 1390
getencoder() (於 codecs 模組中), 195
getencoding() (於 locale 模組中), 1573
getEncoding() (xml.sax.xmlreader.InputSource 的方法), 1410
getEntityResolver() (xml.sax.xmlreader.XMLReader 的方法), 1409
getenv() (於 os 模組中), 671
getenvb() (於 os 模組中), 671
getErrorHandler() (xml.sax.xmlreader.XMLReader 的方法), 1409
geteuid() (於 os 模組中), 671
getEvent() (xml.dom.pulldom.DOMEventStream 的方法), 1399
getEventCategory() (logging.handlers.NTEventLogHandler 的方法), 799
getEventType() (logging.handlers.NTEventLogHandler 的方法), 799
getException() (xml.sax.SAXException 的方法), 1401
getFeature() (xml.sax.xmlreader.XMLReader 的方法), 1409
getfile() (於 inspect 模組中), 2021
getFilesToDelete() (logging.handlers.TimedRotatingFileHandler 的方法), 795
getfilesystemencodeerrors() (於 sys 模組中), 1930
getfilesystemencoding() (於 sys 模組中), 1929
getfirstweekday() (calendar.Calendar 的方法), 256
getfloat() (configparser.ConfigParser 的方法), 641
getfqdn() (於 socket 模組中), 1187
getframeinfo() (於 inspect 模組中), 2030
getframerate() (wave.Wave_read 的方法), 1558
getfullargspec() (於 inspect 模組中), 2027
getgeneratorlocals() (於 inspect 模組中), 2032
getgeneratorstate() (於 inspect 模組中), 2032
getgid() (於 os 模組中), 671
getgrall() (於 grp 模組中), 2198
getgrgid() (於 grp 模組中), 2198
getgrnam() (於 grp 模組中), 2198
getgrouplist() (於 os 模組中), 671
getgroups() (於 os 模組中), 671
getHandlerByName() (於 logging 模組中), 775
getHandlerNames() (於 logging 模組中), 775
getheader() (http.client.HTTPResponse 的方法), 1471
getheaders() (http.client.HTTPResponse 的方法), 1471
gethostbyaddr() (於 socket 模組中), 1187
gethostbyaddr() (於 socket 模組), 676
gethostbyname() (於 socket 模組中), 1187
gethostbyname_ex() (於 socket 模組中), 1187
gethostname() (於 socket 模組中), 1187
gethostname() (於 socket 模組), 676
getincrementaldecoder() (於 codecs 模組中), 195
getincrementalencoder() (於 codecs 模組中), 195
getinfo() (zipfile.ZipFile 的方法), 594
getinnerframes() (於 inspect 模組中), 2030
GetInputContext() (xml.parsers.expat.xmlparser 的方法), 1413
getint() (configparser.ConfigParser 的方法), 641
getitem() (於 operator 模組中), 438
getitimer() (於 signal 模組中), 1252
getkey() (curses.window 的方法), 938
GetLastError() (於 ctypes 模組中), 844
getLength() (xml.sax.xmlreader.Attributes 的方法), 1411
getLevelName() (於 logging 模組中), 775
getLevelNamesMapping() (於 logging 模組中), 775
getlimit() (sqlite3.Connection 的方法), 545
getline() (於 linecache 模組中), 492
getLineNumber() (xml.sax.xmlreader.Locator 的方法), 1410
getloadavg() (於 os 模組中), 731
getlocale() (於 locale 模組中), 1573
getLogger() (於 logging 模組中), 773
getLoggerClass() (於 logging 模組中), 773
getlogin() (於 os 模組中), 672
getLogRecordFactory() (於 logging 模組中), 774
getMandatoryRelease() (__future__._Feature 的方法), 2012
getmark() (wave.Wave_read 的方法), 1558
getmarkers() (wave.Wave_read 的方法), 1558
getmaxyx() (curses.window 的方法), 938
getmember() (tarfile.TarFile 的方法), 607
getmembers() (tarfile.TarFile 的方法), 607
getmembers() (於 inspect 模組中), 2018

getmembers_static() (於 inspect 模組中), 2018
getMessage() (logging.LogRecord 的方法), 770
getMessage() (xml.sax.SAXException 的方法), 1401
getMessageID() (logging.handlers.NTEventLogHandler 的方法), 800
getmodule() (於 inspect 模組中), 2021
getmodulename() (於 inspect 模組中), 2018
getmouse() (於 curses 模組中), 930
getmro() (於 inspect 模組中), 2028
getmtime() (於 os.path 模組中), 470
getName() (threading.Thread 的方法), 973
getNameByQName() (xml.sax.xmlreader.AttributesNS 的方法), 1411
getnameinfo() (asyncio.loop 的方法), 1133
getnameinfo() (於 socket 模組中), 1188
getnames() (tarfile.TarFile 的方法), 607
getNames() (xml.sax.xmlreader.Attributes 的方法), 1411
getnchannels() (wave.Wave_read 的方法), 1558
getnframes() (wave.Wave_read 的方法), 1558
getnode() (於 uuid 模組中), 1499
getobjects() (於 sys 模組中), 1931
getopt
    module, 2215
getopt() (於 getopt 模組中), 2215
GetoptError, 2216
getOptionalRelease() (__future__._Feature 的方法), 2011
getouterframes() (於 inspect 模組中), 2030
getoutput() (於 subprocess 模組中), 1061
getpagesize() (於 resource 模組中), 2209
getparams() (wave.Wave_read 的方法), 1558
getparyx() (curses.window 的方法), 938
getpass
    module, 925
getpass() (於 getpass 模組中), 925
GetPassWarning, 925
getpeercert() (ssl.SSLSocket 的方法), 1216
getpeername() (socket.socket 的方法), 1193
getpen() (於 turtle 模組中), 1651
getpgid() (於 os 模組中), 672
getpgrp() (於 os 模組中), 672
getpid() (於 os 模組中), 672
getpos() (html.parser.HTMLParser 的方法), 1361
getppid() (於 os 模組中), 672
getpreferredencoding() (於 locale 模組中), 1573
getpriority() (於 os 模組中), 672
getprofile() (於 sys 模組中), 1931
getprofile() (於 threading 模組中), 968
getProperty() (xml.sax.xmlreader.XMLReader 的方法), 1409
getprotobyname() (於 socket 模組中), 1188
getproxies() (於 urllib.request 模組中), 1437
getPublicId() (xml.sax.xmlreader.InputSource 的方法), 1410
getPublicId() (xml.sax.xmlreader.Locator 的方法), 1410
getpwall() (於 pwd 模組中), 2198
getpwnam() (於 pwd 模組中), 2198
getpwuid() (於 pwd 模組中), 2198
getQNameByName() (xml.sax.xmlreader.AttributesNS 的方法), 1411
getQNames() (xml.sax.xmlreader.AttributesNS 的方法), 1411
getquota() (imaplib.IMAP4 的方法), 1485
getquotaroot() (imaplib.IMAP4 的方法), 1485
getrandbits() (random.Random 的方法), 389
getrandbits() (於 random 模組中), 386
getrandom() (於 os 模組中), 733
getreader() (於 codecs 模組中), 195
getrecursionlimit() (於 sys 模組中), 1930
getrefcount() (於 sys 模組中), 1930
GetReparseDeferralEnabled() (xml.parsers.expat.xmlparser 的方法), 1413
getresgid() (於 os 模組中), 673
getresponse() (http.client.HTTPConnection 的方法), 1469
getresuid() (於 os 模組中), 673
getrlimit() (於 resource 模組中), 2206
getroot() (xml.etree.ElementTree.ElementTree 的方法), 1379
getrusage() (於 resource 模組中), 2208
getsampwidth() (wave.Wave_read 的方法), 1558
getscreen() (於 turtle 模組中), 1651
getservbyname() (於 socket 模組中), 1188
getservbyport() (於 socket 模組中), 1188
GetSetDescriptorType (於 types 模組中), 307
getshapes() (於 turtle 模組中), 1657
getsid() (於 os 模組中), 675
getsignal() (於 signal 模組中), 1251
getsitepackages() (於 site 模組中), 2045
getsize() (於 os.path 模組中), 470
getsizeof() (於 sys 模組中), 1930
getsockname() (socket.socket 的方法), 1193
getsockopt() (socket.socket 的方法), 1193
getsource() (於 inspect 模組中), 2021
getsourcefile() (於 inspect 模組中), 2021
getsourcelines() (於 inspect 模組中), 2021
getstate() (codecs.IncrementalDecoder 的方法), 201
getstate() (codecs.IncrementalEncoder 的方法), 200
getstate() (random.Random 的方法), 388
getstate() (於 random 模組中), 385
getstatusoutput() (於 subprocess 模組中), 1061
getstr() (curses.window 的方法), 938
getSubject() (logging.handlers.SMTPHandler 的方法), 800
getswitchinterval() (於 sys 模組中), 1930
getSystemId() (xml.sax.xmlreader.InputSource 的方法), 1410
getSystemId() (xml.sax.xmlreader.Locator 的方法), 1410
getsyx() (於 curses 模組中), 931
gettarinfo() (tarfile.TarFile 的方法), 609
gettempdir() (於 tempfile 模組中), 486

gettempdirb() (於 *tempfile* 模組中), 486
gettempprefix() (於 *tempfile* 模組中), 486
gettempprefixb() (於 *tempfile* 模組中), 486
getTestCaseNames() (*unittest.TestLoader* 的方法), 1768
gettext
    module, 1561
gettext() (*gettext.GNUTranslations* 的方法), 1565
gettext() (*gettext.NullTranslations* 的方法), 1563
gettext() (於 *gettext* 模組中), 1562
gettext() (於 *locale* 模組中), 1577
gettimeout() (*socket.socket* 的方法), 1193
gettrace() (於 *sys* 模組中), 1931
gettrace() (於 *threading* 模組中), 968
getturtle() (於 *turtle* 模組中), 1651
getType() (*xml.sax.xmlreader.Attributes* 的方法), 1411
getuid() (於 *os* 模組中), 673
getunicodeinternedsize() (於 *sys* 模組中), 1929
geturl() (*http.client.HTTPResponse* 的方法), 1472
geturl() (*urllib.parse.urllib.parse.SplitResult* 的方法), 1458
geturl() (*urllib.response.addinfourl* 的方法), 1452
getuser() (於 *getpass* 模組中), 925
getuserbase() (於 *site* 模組中), 2045
getusersitepackages() (於 *site* 模組中), 2045
getvalue() (*io.BytesIO* 的方法), 742
getvalue() (*io.StringIO* 的方法), 746
getValue() (*xml.sax.xmlreader.Attributes* 的方法), 1411
getValueByQName() (*xml.sax.xmlreader.AttributesNS* 的方法), 1411
getwch() (於 *msvcrt* 模組中), 2178
getwche() (於 *msvcrt* 模組中), 2178
getweakrefcount() (於 *weakref* 模組中), 298
getweakrefs() (於 *weakref* 模組中), 298
getwelcome() (*ftplib.FTP* 的方法), 1475
getwelcome() (*poplib.POP3* 的方法), 1481
getwin() (於 *curses* 模組中), 931
getwindowsversion() (於 *sys* 模組中), 1931
getwriter() (於 *codecs* 模組中), 195
getxattr() (於 *os* 模組中), 715
getyx() (*curses.window* 的方法), 939
gid (*tarfile.TarInfo* 的屬性), 611
GIL, **2234**
gil (*sys.flags* 的屬性), 1926
glob
    module, 488
    module（模組）, 490
glob() (*pathlib.Path* 的方法), 460
glob() (於 *glob* 模組中), 488
Global (*ast* 中的類🄵), 2123
global interpreter lock（全域直譯器鎖）, **2234**
global_enum() (於 *enum* 模組中), 333
globals()
    built-in function, 17
globs (*doctest.DocTest* 的屬性), 1738
gmtime() (於 *time* 模組中), 750
gname (*tarfile.TarInfo* 的屬性), 611
GNOME, 1565
GNU_FORMAT (於 *tarfile* 模組中), 606
gnu_getopt() (於 *getopt* 模組中), 2216
GNUTranslations (*gettext* 中的類🄵), 1565
GNUTYPE_LONGLINK (於 *tarfile* 模組中), 605
GNUTYPE_LONGNAME (於 *tarfile* 模組中), 605
GNUTYPE_SPARSE (於 *tarfile* 模組中), 605
go() (*tkinter.filedialog.FileDialog* 的方法), 1594
got (*doctest.DocTestFailure* 的屬性), 1745
goto() (於 *turtle* 模組中), 1636
grantpt() (於 *os* 模組中), 680
Graphical User Interface（圖形使用者介面）, 1579
graphlib
    module, 334
GREATER (於 *token* 模組中), 2136
GREATEREQUAL (於 *token* 模組中), 2137
greedy (*compression.zstd.Strategy* 的屬性), 572
Greenwich Mean Time（格林威治標准時間）, 748
GRND_NONBLOCK (於 *os* 模組中), 733
GRND_RANDOM (於 *os* 模組中), 733
Group (*email.headerregistry* 中的類🄵), 1291
group() (*pathlib.Path* 的方法), 465
group() (*re.Match* 的方法), 153
groupby() (於 *itertools* 模組中), 414
groupdict() (*re.Match* 的方法), 155
groupindex (*re.Pattern* 的屬性), 153
groups (*email.headerregistry.AddressHeader* 的屬性), 1288
groups (*re.Pattern* 的屬性), 153
groups() (*re.Match* 的方法), 154
grp
    module, 2198
GS (於 *curses.ascii* 模組中), 956
Gt (*ast* 中的類🄵), 2103
gt() (於 *operator* 模組中), 435
GtE (*ast* 中的類🄵), 2103
guess_all_extensions() (*mimetypes.MimeTypes* 的方法), 1350
guess_all_extensions() (於 *mimetypes* 模組中), 1349
guess_extension() (*mimetypes.MimeTypes* 的方法), 1350
guess_extension() (於 *mimetypes* 模組中), 1349
guess_file_type() (*mimetypes.MimeTypes* 的方法), 1350
guess_file_type() (於 *mimetypes* 模組中), 1348
guess_scheme() (於 *wsgiref.util* 模組中), 1426
guess_type() (*mimetypes.MimeTypes* 的方法), 1350
guess_type() (於 *mimetypes* 模組中), 1348
GUI, 1579
gzip
    module, 577
gzip 命令列選項
    --best, 581
    -d, 581
    --decompress, 581

    --fast, 581
    file, 581
    -h, 581
    --help, 581
GzipFile (*gzip* 中的類🄵), 578

H

-h
    ast 命令列選項, 2129
    calendar 命令列選項, 262
    dis 命令列選項, 2152
    gzip 命令列選項, 581
    idle 命令列選項, 1624
    json 命令列選項, 1329
    mimetypes 命令列選項, 1351
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令列選項, 554
    random 命令列選項, 392
    timeit 命令列選項, 1888
    tokenize 命令列選項, 2141
    uuid 命令列選項, 1501
    zipapp 命令列選項, 1915
halfdelay() (於 *curses* 模組中), 931
Handle (*asyncio* 中的類🄵), 1138
handle() (*http.server.BaseHTTPRequestHandler* 的方法), 1513
handle() (*logging.Handler* 的方法), 767
handle() (*logging.handlers.QueueListener* 的方法), 803
handle() (*logging.Logger* 的方法), 765
handle() (*logging.NullHandler* 的方法), 792
handle() (*socketserver.BaseRequestHandler* 的方法), 1507
handle() (*wsgiref.simple_server.WSGIRequestHandler* 的方法), 1429
handle_charref() (*html.parser.HTMLParser* 的方法), 1361
handle_comment() (*html.parser.HTMLParser* 的方法), 1361
handle_data() (*html.parser.HTMLParser* 的方法), 1361
handle_decl() (*html.parser.HTMLParser* 的方法), 1362
handle_defect() (*email.policy.Policy* 的方法), 1280
handle_endtag() (*html.parser.HTMLParser* 的方法), 1361
handle_entityref() (*html.parser.HTMLParser* 的方法), 1361
handle_error() (*socketserver.BaseServer* 的方法), 1506
handle_expect_100() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
handle_one_request() (*http.server.BaseHTTPRequestHandler* 的方法), 1513
handle_pi() (*html.parser.HTMLParser* 的方法), 1362
handle_request() (*socketserver.BaseServer* 的方法), 1505
handle_request() (*xmlrpc.server.CGIXMLRPCRequestHandler* 的方法), 1542
handle_startendtag() (*html.parser.HTMLParser* 的方法), 1361
handle_starttag() (*html.parser.HTMLParser* 的方法), 1361
handle_timeout() (*socketserver.BaseServer* 的方法), 1506
handleError() (*logging.Handler* 的方法), 767
handleError() (*logging.handlers.SocketHandler* 的方法), 796
Handler (*logging* 中的類🄵), 766
handlers (*logging.Logger* 的屬性), 762
Handlers (*signal* 中的類🄵), 1248
hardlink_to() (*pathlib.Path* 的方法), 463
--hardlink-dupes
    compileall 命令列選項, 2148
harmonic_mean() (於 *statistics* 模組中), 396
HAS_ALPN (於 *ssl* 模組中), 1212
has_children() (*symtable.SymbolTable* 的方法), 2131
has_colors() (於 *curses* 模組中), 931
has_default() (*typing.ParamSpec* 的方法), 1696
has_default() (*typing.TypeVar* 的方法), 1693
has_default() (*typing.TypeVarTuple* 的方法), 1695
has_dualstack_ipv6() (於 *socket* 模組中), 1185
HAS_ECDH (於 *ssl* 模組中), 1212
has_extended_color_support() (於 *curses* 模組中), 931
has_extn() (*smtplib.SMTP* 的方法), 1493
has_header() (*csv.Sniffer* 的方法), 622
has_header() (*urllib.request.Request* 的方法), 1442
has_ic() (於 *curses* 模組中), 931
has_il() (於 *curses* 模組中), 931
has_ipv6 (於 *socket* 模組中), 1181
has_key() (於 *curses* 模組中), 931
has_location (*importlib.machinery.ModuleSpec* 的屬性), 2073
HAS_NEVER_CHECK_COMMON_NAME (於 *ssl* 模組中), 1212
has_nonstandard_attr() (*http.cookiejar.Cookie* 的方法), 1529
HAS_NPN (於 *ssl* 模組中), 1212
has_option() (*configparser.ConfigParser* 的方法), 640
has_option() (*optparse.OptionParser* 的方法), 917
HAS_PHA (於 *ssl* 模組中), 1212
HAS_PSK (於 *ssl* 模組中), 1212
has_section() (*configparser.ConfigParser* 的方法), 640
HAS_SNI (於 *ssl* 模組中), 1212
HAS_SSLv2 (於 *ssl* 模組中), 1212
HAS_SSLv3 (於 *ssl* 模組中), 1212
has_ticket (*ssl.SSLSession* 的屬性), 1234
HAS_TLSv1 (於 *ssl* 模組中), 1212

HAS_TLSv1_1 (於 *ssl* 模組中), 1212
HAS_TLSv1_2 (於 *ssl* 模組中), 1212
HAS_TLSv1_3 (於 *ssl* 模組中), 1212
hasarg (於 *dis* 模組中), 2173
hasattr()
    built-in function, 17
hasAttribute() (*xml.dom.Element* 的方法), 1390
hasAttributeNS() (*xml.dom.Element* 的方法), 1390
hasAttributes() (*xml.dom.Node* 的方法), 1387
hasChildNodes() (*xml.dom.Node* 的方法), 1387
hascompare (於 *dis* 模組中), 2173
hasconst (於 *dis* 模組中), 2173
hasexc (於 *dis* 模組中), 2173
hasFeature() (*xml.dom.DOMImplementation* 的方法), 1386
hasfree (於 *dis* 模組中), 2173
hash()
    built-in function, 17
hash-based pyc（雜🄵架構的 pyc）, **2234**
hash_bits (*sys.hash_info* 的屬性), 1933
hash_info (於 *sys* 模組中), 1932
hash_log (*compression.zstd.CompressionParameter* 的屬性), 569
hash_randomization (*sys.flags* 的屬性), 1926
Hashable (*collections.abc* 中的類🄵), 283
Hashable (*typing* 中的類🄵), 1719
hashable（可雜🄵的）, **2234**
hasHandlers() (*logging.Logger* 的方法), 765
hash.block_size (於 *hashlib* 模組中), 653
hash.digest_size (於 *hashlib* 模組中), 653
hashlib
    module, 651
hash（雜🄵）
    built-in function（🄵建函式）, 48
hasjabs (於 *dis* 模組中), 2173
hasjrel (於 *dis* 模組中), 2173
hasjump (於 *dis* 模組中), 2173
haslocal (於 *dis* 模組中), 2173
hasname (於 *dis* 模組中), 2173
HAVE_ARGUMENT (*opcode*), 2171
HAVE_CONTEXTVAR (於 *decimal* 模組中), 371
HAVE_DOCSTRINGS (於 *test.support* 模組中), 1838
HAVE_THREADS (於 *decimal* 模組中), 371
HCI_CHANNEL_CONTROL (於 *socket* 模組中), 1182
HCI_CHANNEL_LOGGING (於 *socket* 模組中), 1182
HCI_CHANNEL_MONITOR (於 *socket* 模組中), 1182
HCI_CHANNEL_RAW (於 *socket* 模組中), 1182
HCI_CHANNEL_USER (於 *socket* 模組中), 1182
HCI_DATA_DIR (於 *socket* 模組中), 1182
HCI_DEV_NONE (於 *socket* 模組中), 1182
HCI_FILTER (於 *socket* 模組中), 1182
HCI_TIME_STAMP (於 *socket* 模組中), 1182
Header (*email.header* 中的類🄵), 1311
header_encode() (*email.charset.Charset* 的方法), 1314
header_encode_lines() (*email.charset.Charset* 的方法), 1314
header_encoding (*email.charset.Charset* 的屬性), 1314
header_factory (*email.policy.EmailPolicy* 的屬性), 1282
header_fetch_parse() (*email.policy.Compat32* 的方法), 1284
header_fetch_parse() (*email.policy.EmailPolicy* 的方法), 1283
header_fetch_parse() (*email.policy.Policy* 的方法), 1281
header_items() (*urllib.request.Request* 的方法), 1442
header_max_count() (*email.policy.EmailPolicy* 的方法), 1282
header_max_count() (*email.policy.Policy* 的方法), 1281
header_offset (*zipfile.ZipInfo* 的屬性), 600
header_source_parse() (*email.policy.Compat32* 的方法), 1284
header_source_parse() (*email.policy.EmailPolicy* 的方法), 1282
header_source_parse() (*email.policy.Policy* 的方法), 1281
header_store_parse() (*email.policy.Compat32* 的方法), 1284
header_store_parse() (*email.policy.EmailPolicy* 的方法), 1283
header_store_parse() (*email.policy.Policy* 的方法), 1281
HeaderDefect, 1285
HeaderError, 604
HeaderParseError, 1285
HeaderParser (*email.parser* 中的類🄵), 1274
HeaderRegistry (*email.headerregistry* 中的類🄵), 1289
headers (*http.client.HTTPResponse* 的屬性), 1472
headers (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
headers (*urllib.error.HTTPError* 的屬性), 1461
headers (*urllib.response.addinfourl* 的屬性), 1452
Headers (*wsgiref.headers* 中的類🄵), 1428
headers (*xmlrpc.client.ProtocolError* 的屬性), 1535
headers（標頭）
    MIME, 1348
HeaderWriteError, 1285
heading() (*tkinter.ttk.Treeview* 的方法), 1610
heading() (於 *turtle* 模組中), 1641
heapify() (於 *heapq* 模組中), 287
heapify_max() (於 *heapq* 模組中), 287
heapmin() (於 *msvcrt* 模組中), 2178
heappop() (於 *heapq* 模組中), 286
heappop_max() (於 *heapq* 模組中), 287
heappush() (於 *heapq* 模組中), 286
heappush_max() (於 *heapq* 模組中), 287
heappushpop() (於 *heapq* 模組中), 286
heappushpop_max() (於 *heapq* 模組中), 287
heapq
    module, 286

heapreplace() (於 *heapq* 模組中), 287
heapreplace_max() (於 *heapq* 模組中), 287
helo() (*smtplib.SMTP* 的方法), 1493
--help
    ast 命令列選項, 2129
    calendar 命令列選項, 262
    dis 命令列選項, 2152
    gzip 命令列選項, 581
    json 命令列選項, 1329
    mimetypes 命令列選項, 1351
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令列選項, 554
    random 命令列選項, 392
    timeit 命令列選項, 1888
    tokenize 命令列選項, 2141
    trace 命令列選項, 1891
    uuid 命令列選項, 1501
    zipapp 命令列選項, 1915
help (*optparse.Option* 的屬性), 912
help (*pdb command*), 1872
help()
    built-in function, 17
help（幫助）
    online（🄵上）, 1720
herror, 1177
hex (*uuid.UUID* 的屬性), 1498
hex()
    built-in function, 18
hex() (*bytearray* 的方法), 68
hex() (*bytes* 的方法), 67
hex() (*float* 的方法), 43
hex() (*memoryview* 的方法), 84
hexadecimal（十六進位）
    literals（字面值）, 38
hexdigest() (*hashlib.hash* 的方法), 653
hexdigest() (*hashlib.shake* 的方法), 654
hexdigest() (*hmac.HMAC* 的方法), 663
hexdigits (於 *string* 模組中), 123
hexlify() (於 *binascii* 模組中), 1357
hexversion (於 *sys* 模組中), 1933
hidden() (*curses.panel.Panel* 的方法), 959
hide() (*curses.panel.Panel* 的方法), 959
hide() (*tkinter.ttk.Notebook* 的方法), 1605
hide_cookie2 (*http.cookiejar.CookiePolicy* 的屬性), 1526
hideturtle() (於 *turtle* 模組中), 1646
HierarchyRequestErr, 1392
HIGH_PRIORITY_CLASS (於 *subprocess* 模組中), 1057
HIGHEST_PROTOCOL (於 *pickle* 模組中), 507
hits (*bdb.Breakpoint* 的屬性), 1860
HKEY_CLASSES_ROOT (於 *winreg* 模組中), 2185
HKEY_CURRENT_CONFIG (於 *winreg* 模組中), 2186
HKEY_CURRENT_USER (於 *winreg* 模組中), 2185
HKEY_DYN_DATA (於 *winreg* 模組中), 2186
HKEY_LOCAL_MACHINE (於 *winreg* 模組中), 2185
HKEY_PERFORMANCE_DATA (於 *winreg* 模組中), 2185
HKEY_USERS (於 *winreg* 模組中), 2185
hline() (*curses.window* 的方法), 939
hls_to_rgb() (於 *colorsys* 模組中), 1560
hmac
    module, 662
HMAC (*hmac* 中的類🄵), 662
HOME, 470, 1581
home() (*pathlib.Path* 的類🄵方法), 455
home() (於 *turtle* 模組中), 1638
HOMEDRIVE, 470
HOMEPATH, 470
hook_compressed() (於 *fileinput* 模組中), 928
hook_encoded() (於 *fileinput* 模組中), 928
host (*urllib.request.Request* 的屬性), 1441
hostmask (*ipaddress.IPv4Network* 的屬性), 1550
hostmask (*ipaddress.IPv6Network* 的屬性), 1553
hostname_checks_common_name (*ssl.SSLContext* 的屬性), 1225
hosts (*netrc.netrc* 的屬性), 647
hosts() (*ipaddress.IPv4Network* 的方法), 1550
hosts() (*ipaddress.IPv6Network* 的方法), 1553
hour (*datetime.datetime* 的屬性), 228
hour (*datetime.time* 的屬性), 236
HRESULT (*ctypes* 中的類🄵), 849
hresult (*ctypes.COMError* 的屬性), 853
hStdError (*subprocess.STARTUPINFO* 的屬性), 1055
hStdInput (*subprocess.STARTUPINFO* 的屬性), 1055
hStdOutput (*subprocess.STARTUPINFO* 的屬性), 1055
hsv_to_rgb() (於 *colorsys* 模組中), 1560
HT (於 *curses.ascii* 模組中), 955
ht() (於 *turtle* 模組中), 1646
HTML, 1360, 1451
html
    module, 1359
html5 (於 *html.entities* 模組中), 1364
HTMLCalendar (*calendar* 中的類🄵), 258
HtmlDiff (*difflib* 中的類🄵), 161
html.entities
    module, 1364
html.parser
    module, 1360
HTMLParser (*html.parser* 中的類🄵), 1360
htonl() (於 *socket* 模組中), 1188
htons() (於 *socket* 模組中), 1188
HTTP
    http.client（標准模組）, 1466
    http（標准模組）, 1463
    protocol（協定）, 1451, 1463, 1466, 1511
http
    module, 1463
HTTP (於 *email.policy* 模組中), 1283
http_error_301() (*urllib.request.HTTPRedirectHandler* 的方法), 1445
http_error_302() (*urllib.request.HTTPRedirectHandler* 的方法), 1445
http_error_303() (*urllib.request.HTTPRedirectHandler* 的方

法), 1445
http_error_307() (urllib.request.HTTPRedirectHandler 的方法), 1445
http_error_308() (urllib.request.HTTPRedirectHandler 的方法), 1445
http_error_401() (urllib.request.HTTPBasicAuthHandler 的方法), 1446
http_error_401() (urllib.request.HTTPDigestAuthHandler 的方法), 1447
http_error_407() (urllib.request.ProxyBasicAuthHandler 的方法), 1446
http_error_407() (urllib.request.ProxyDigestAuthHandler 的方法), 1447
http_error_auth_reqed() (urllib.request.AbstractBasicAuthHandler 的方法), 1446
http_error_auth_reqed() (urllib.request.AbstractDigestAuthHandler 的方法), 1446
http_error_default() (urllib.request.BaseHandler 的方法), 1444
http_open() (urllib.request.HTTPHandler 的方法), 1447
HTTP_PORT (於 http.client 模組中), 1468
http_response() (urllib.request.HTTPErrorProcessor 的方法), 1448
http_version (wsgiref.handlers.BaseHandler 的屬性), 1433
HTTPBasicAuthHandler (urllib.request 中的類🄵), 1440
http.client
    module, 1466
HTTPConnection (http.client 中的類🄵), 1466
http.cookiejar
    module, 1522
HTTPCookieProcessor (urllib.request 中的類🄵), 1439
http.cookies
    module, 1518
httpd, 1511
HTTPDefaultErrorHandler (urllib.request 中的類🄵), 1439
HTTPDigestAuthHandler (urllib.request 中的類🄵), 1440
HTTPError, 1461
HTTPErrorProcessor (urllib.request 中的類🄵), 1440
HTTPException, 1468
HTTPHandler (logging.handlers 中的類🄵), 801
HTTPHandler (urllib.request 中的類🄵), 1440
HTTPMessage (http.client 中的類🄵), 1473
HTTPMethod (http 中的類🄵), 1465
httponly (http.cookies.Morsel 的屬性), 1520
HTTPPasswordMgr (urllib.request 中的類🄵), 1439
HTTPPasswordMgrWithDefaultRealm (urllib.request 中的類🄵), 1439
HTTPPasswordMgrWithPriorAuth (urllib.request 中的類🄵), 1439
HTTPRedirectHandler (urllib.request 中的類🄵), 1439
HTTPResponse (http.client 中的類🄵), 1467
https_open() (urllib.request.HTTPSHandler 的方法), 1447
HTTPS_PORT (於 http.client 模組中), 1468
https_response() (urllib.request.HTTPErrorProcessor 的方法), 1448
HTTPSConnection (http.client 中的類🄵), 1467
http.server
    module, 1511
    security（安全）, 1518
HTTPServer (http.server 中的類🄵), 1511
http.server 命令列選項
    -b, 1517
    --bind, 1517
    --cgi, 1517
    -d, 1517
    --directory, 1517
    -p, 1517
    port, 1517
    --protocol, 1517
    --tls-cert, 1518
    --tls-key, 1518
    --tls-password-file, 1518
HTTPSHandler (urllib.request 中的類🄵), 1440
HTTPSServer (http.server 中的類🄵), 1511
HTTPStatus (http 中的類🄵), 1463
HV_GUID_BROADCAST (於 socket 模組中), 1183
HV_GUID_CHILDREN (於 socket 模組中), 1183
HV_GUID_LOOPBACK (於 socket 模組中), 1183
HV_GUID_PARENT (於 socket 模組中), 1183
HV_GUID_WILDCARD (於 socket 模組中), 1183
HV_GUID_ZERO (於 socket 模組中), 1183
HV_PROTOCOL_RAW (於 socket 模組中), 1183
HVSOCKET_ADDRESS_FLAG_PASSTHRU (於 socket 模組中), 1183
HVSOCKET_CONNECT_TIMEOUT (於 socket 模組中), 1183
HVSOCKET_CONNECT_TIMEOUT_MAX (於 socket 模組中), 1183
HVSOCKET_CONNECTED_SUSPEND (於 socket 模組中), 1183
hypot() (於 math 模組中), 346

I

-i
    ast 命令列選項, 2129
    compileall 命令列選項, 2148
    idle 命令列選項, 1624
    random 命令列選項, 392

I (於 re 模組中), 146
I/O control (I/O 控制)
    buffering (緩衝), 24, 1194
    POSIX, 2199
    tty, 2199
    UNIX, 2203
iadd() (於 operator 模組中), 441
iand() (於 operator 模組中), 441
iconcat() (於 operator 模組中), 441
id (concurrent.interpreters.Interpreter 的屬性), 1042
id (concurrent.interpreters.Queue 的屬性), 1043
id (ssl.SSLSession 的屬性), 1234
id()
    built-in function, 18
id() (unittest.TestCase 的方法), 1764
idcok() (curses.window 的方法), 939
ident (select.kevent 的屬性), 1242
ident (threading.Thread 的屬性), 973
identchars (cmd.Cmd 的屬性), 961
identify() (tkinter.ttk.Notebook 的方法), 1605
identify() (tkinter.ttk.Treeview 的方法), 1610
identify() (tkinter.ttk.Widget 的方法), 1601
identify_column() (tkinter.ttk.Treeview 的方法), 1610
identify_element() (tkinter.ttk.Treeview 的方法), 1611
identify_region() (tkinter.ttk.Treeview 的方法), 1610
identify_row() (tkinter.ttk.Treeview 的方法), 1610
IDLE, 1617, **2234**
idle 命令列選項
    -, 1624
    -c, 1624
    -d, 1624
    -e, 1624
    -h, 1624
    -i, 1624
    -r, 1624
    -s, 1624
    -t, 1624
idle() (imaplib.IMAP4 的方法), 1485
IDLE_PRIORITY_CLASS (於 subprocess 模組中), 1057
idlelib
    module, 1628
IDLESTARTUP, 1624
idlok() (curses.window 的方法), 939
IEEE_CONTEXT_MAX_BITS (於 decimal 模組中), 371
IEEEContext() (於 decimal 模組中), 365
if
    statement (陳述式), 37
If (ast 中的類🄵), 2110
if_indextoname() (於 socket 模組中), 1191
if_nameindex() (於 socket 模組中), 1190
if_nametoindex() (於 socket 模組中), 1190
IfExp (ast 中的類🄵), 2103
ifloordiv() (於 operator 模組中), 441
iglob() (於 glob 模組中), 489
ignorableWhitespace()
    (xml.sax.handler.ContentHandler 的方法), 1405
ignore
    error handler's name (錯誤處理器名稱), 197
ignore (bdb.Breakpoint 的屬性), 1860
ignore (pdb command), 1873
IGNORE (於 tkinter.messagebox 模組中), 1597
ignore_environment (sys.flags 的屬性), 1926
ignore_errors() (於 codecs 模組中), 198
IGNORE_EXCEPTION_DETAIL (於 doctest 模組中), 1731
ignore_patterns() (於 shutil 模組中), 495
ignore_warnings() (於 test.support.warnings_helper 模組中), 1852
IGNORECASE (於 re 模組中), 146
--ignore-dir
    trace 命令列選項, 1892
--ignore-module
    trace 命令列選項, 1892
IISCGIHandler (wsgiref.handlers 中的類🄵), 1430
IllegalMonthError, 261
IllegalWeekdayError, 261
ilshift() (於 operator 模組中), 441
imag (numbers.Complex 的屬性), 337
imag (sys.hash_info 的屬性), 1933
imap() (multiprocessing.pool.Pool 的方法), 1012
IMAP4
    protocol (協定), 1483
IMAP4 (imaplib 中的類🄵), 1483
IMAP4_SSL
    protocol (協定), 1483
IMAP4_SSL (imaplib 中的類🄵), 1483
IMAP4_stream
    protocol (協定), 1483
IMAP4_stream (imaplib 中的類🄵), 1484
IMAP4.abort, 1483
IMAP4.error, 1483
IMAP4.readonly, 1483
imap_unordered() (multiprocessing.pool.Pool 的方法), 1012
imaplib
    module, 1483
imatmul() (於 operator 模組中), 441
imghdr
    module, 2221
immedok() (curses.window 的方法), 939
immortal (不滅), **2234**
immutable (不可變物件), **2235**
immutable (不可變)
    sequence (序列) type (型🄵), 48
imod() (於 operator 模組中), 441
imp
    module, 2221
impl_detail() (於 test.support 模組中), 1842
implementation (於 sys 模組中), 1933
Import (ast 中的類🄵), 2109
import path (引入路徑), **2235**

import_fresh_module() (於 test.support.import_helper 模組中), 1851
IMPORT_FROM (opcode), 2166
import_module() (於 importlib 模組中), 2060
import_module() (於 test.support.import_helper 模組中), 1851
IMPORT_NAME (opcode), 2165
ImportError, 111
importer（引入器）, 2235
ImportFrom (ast 中的類🄵), 2109
importing（引入）, 2235
importlib
    module, 2059
importlib.abc
    module, 2062
importlib.machinery
    module, 2068
importlib.metadata
    module, 2083
importlib.resources
    module, 2079
importlib.resources.abc
    module, 2082
importlib.util
    module, 2073
ImportWarning, 118
import（引入）
    statement（陳述式）, 32, 2043
ImproperConnectionState, 1468
imul() (於 operator 模組中), 441
in
    operator（運算子）, 38, 46
In (ast 中的類🄵), 2103
in_dll() (ctypes._CData 的方法), 846
in_table_a1() (於 stringprep 模組中), 179
in_table_b1() (於 stringprep 模組中), 179
in_table_c3() (於 stringprep 模組中), 179
in_table_c4() (於 stringprep 模組中), 179
in_table_c5() (於 stringprep 模組中), 179
in_table_c6() (於 stringprep 模組中), 180
in_table_c7() (於 stringprep 模組中), 180
in_table_c8() (於 stringprep 模組中), 180
in_table_c9() (於 stringprep 模組中), 180
in_table_c11() (於 stringprep 模組中), 179
in_table_c11_c12() (於 stringprep 模組中), 179
in_table_c12() (於 stringprep 模組中), 179
in_table_c21() (於 stringprep 模組中), 179
in_table_c21_c22() (於 stringprep 模組中), 179
in_table_c22() (於 stringprep 模組中), 179
in_table_d1() (於 stringprep 模組中), 180
in_table_d2() (於 stringprep 模組中), 180
in_transaction (sqlite3.Connection 的屬性), 547
inch() (curses.window 的方法), 939
include() (於 xml.etree.ElementInclude 模組中), 1376
--include-attributes
    ast 命令列選項, 2129
inclusive (tracemalloc.DomainFilter 的屬性), 1899
inclusive (tracemalloc.Filter 的屬性), 1899
Incomplete, 1358
IncompleteRead, 1468
IncompleteReadError, 1118
increment_lineno() (於 ast 模組中), 2126
IncrementalDecoder (codecs 中的類🄵), 200
incrementaldecoder (codecs.CodecInfo 的屬性), 194
IncrementalEncoder (codecs 中的類🄵), 200
incrementalencoder (codecs.CodecInfo 的屬性), 194
IncrementalNewlineDecoder (io 中的類🄵), 746
IncrementalParser (xml.sax.xmlreader 中的類🄵), 1407
--indent
    ast 命令列選項, 2129
    json 命令列選項, 1329
indent (doctest.Example 的屬性), 1739
indent (reprlib.Repr 的屬性), 317
INDENT (於 token 模組中), 2134
indent() (於 textwrap 模組中), 174
indent() (於 xml.etree.ElementTree 模組中), 1373
IndentationError, 114
--indentlevel
    pickletools 命令列選項, 2174
index (inspect.FrameInfo 的屬性), 2029
index (inspect.Traceback 的屬性), 2030
index() (array.array 的方法), 295
index() (bytearray 的方法), 71
index() (bytes 的方法), 71
index() (collections.abc.ByteString 的方法), 283
index() (collections.deque 的方法), 269
index() (memoryview 的方法), 87
index() (multiprocessing.shared_memory.ShareableList 的方法), 1030
index() (sequence 的方法), 47
index() (str 的方法), 57
index() (tkinter.ttk.Notebook 的方法), 1605
index() (tkinter.ttk.Treeview 的方法), 1611
index() (於 operator 模組中), 436
IndexError, 111
indexOf() (於 operator 模組中), 438
IndexSizeErr, 1392
INDIRECT (inspect.BufferFlags 的屬性), 2034
inet_aton() (於 socket 模組中), 1189
inet_ntoa() (於 socket 模組中), 1189
inet_ntop() (於 socket 模組中), 1189
inet_pton() (於 socket 模組中), 1189
Inexact (decimal 中的類🄵), 372
inf (sys.hash_info 的屬性), 1932
inf (於 cmath 模組中), 352
inf (於 math 模組中), 348
infile
    json 命令列選項, 1328
infile (shlex.shlex 的屬性), 2194
Infinity（無窮）, 16
infj (於 cmath 模組中), 352
--info
    zipapp 命令列選項, 1915

info (*pathlib.Path* 的屬性), 459
INFO (於 *logging* 模組中), 766
INFO (於 *tkinter.messagebox* 模組中), 1597
info() (*dis.Bytecode* 的方法), 2153
info() (*gettext.NullTranslations* 的方法), 1564
info() (*http.client.HTTPResponse* 的方法), 1472
info() (*logging.Logger* 的方法), 764
info() (*urllib.response.addinfourl* 的方法), 1452
info() (於 *logging* 模組中), 774
infolist() (*zipfile.ZipFile* 的方法), 594
.ini
    file（檔案）, 626
ini file（ini 檔案）, 626
init() (於 *mimetypes* 模組中), 1349
init_color() (於 *curses* 模組中), 931
init_pair() (於 *curses* 模組中), 931
inited (於 *mimetypes* 模組中), 1349
initgroups() (於 *os* 模組中), 673
initial_indent (*textwrap.TextWrapper* 的屬性), 175
initscr() (於 *curses* 模組中), 931
InitVar (*dataclasses* 中的類F), 1973
inode() (*os.DirEntry* 的方法), 701
input()
    built-in function, 18
input() (於 *fileinput* 模組中), 926
input_charset (*email.charset.Charset* 的屬性), 1314
input_codec (*email.charset.Charset* 的屬性), 1314
InputSource (*xml.sax.xmlreader* 中的類F), 1408
InputStream (*wsgiref.types* 中的類F), 1433
insch() (*curses.window* 的方法), 939
insdelln() (*curses.window* 的方法), 939
insert() (*array.array* 的方法), 295
insert() (*collections.deque* 的方法), 269
insert() (*sequence* 的方法), 49
insert() (*tkinter.ttk.Notebook* 的方法), 1605
insert() (*tkinter.ttk.Treeview* 的方法), 1611
insert() (*xml.etree.ElementTree.Element* 的方法), 1378
insert_text() (於 *readline* 模組中), 181
insertBefore() (*xml.dom.Node* 的方法), 1388
insertln() (*curses.window* 的方法), 939
insnstr() (*curses.window* 的方法), 939
insort() (於 *bisect* 模組中), 291
insort_left() (於 *bisect* 模組中), 291
insort_right() (於 *bisect* 模組中), 291
inspect
    module, 2016
inspect (*sys.flags* 的屬性), 1926
inspect 命令列選項
    --details, 2035
InspectLoader (*importlib.abc* 中的類F), 2064
insstr() (*curses.window* 的方法), 939
install() (*gettext.NullTranslations* 的方法), 1564
install() (於 *gettext* 模組中), 1563
install_opener() (於 *urllib.request* 模組中), 1436
install_scripts() (*venv.EnvBuilder* 的方法), 1910
installHandler() (於 *unittest* 模組中), 1776
instate() (*tkinter.ttk.Widget* 的方法), 1602
instr() (*curses.window* 的方法), 939
instream (*shlex.shlex* 的屬性), 2194
Instruction (*dis* 中的類F), 2156
INSTRUCTION (*monitoring event*), 1948
Instruction.arg (於 *dis* 模組中), 2156
Instruction.argrepr (於 *dis* 模組中), 2156
Instruction.argval (於 *dis* 模組中), 2156
Instruction.baseopcode (於 *dis* 模組中), 2156
Instruction.baseopname (於 *dis* 模組中), 2156
Instruction.cache_info (於 *dis* 模組中), 2157
Instruction.cache_offset (於 *dis* 模組中), 2156
Instruction.end_offset (於 *dis* 模組中), 2156
Instruction.is_jump_target (於 *dis* 模組中), 2156
Instruction.jump_target (於 *dis* 模組中), 2156
Instruction.line_number (於 *dis* 模組中), 2156
Instruction.offset (於 *dis* 模組中), 2156
Instruction.oparg (於 *dis* 模組中), 2156
Instruction.opcode (於 *dis* 模組中), 2156
Instruction.opname (於 *dis* 模組中), 2156
Instruction.positions (於 *dis* 模組中), 2156
Instruction.start_offset (於 *dis* 模組中), 2156
Instruction.starts_line (於 *dis* 模組中), 2156
int
    built-in function（F建函式）, 38
int (*uuid.UUID* 的屬性), 1499
int (F建類F), 18
Int2AP() (於 *imaplib* 模組中), 1484
int_info (於 *sys* 模組中), 1934
int_max_str_digits (*sys.flags* 的屬性), 1926
--integer
    random 命令列選項, 392
integer（整數）
    literals（字面值）, 38
    object（物件）, 38
    type（型F）, operations on（操作於）, 40
Integral (*numbers* 中的類F), 338
Integrated Development Environment（整合開發環境）, 1617
IntegrityError, 552
IntEnum (*enum* 中的類F), 326
interact (*pdb command*), 1875
interact() (*code.InteractiveConsole* 的方法), 2048
interact() (於 *code* 模組中), 2047
Interactive (*ast* 中的類F), 2097
interactive (*sys.flags* 的屬性), 1926
InteractiveConsole (*code* 中的類F), 2047
InteractiveInterpreter (*code* 中的類F), 2047
interactive（互動的）, **2235**
InterfaceError, 552
intern() (於 *sys* 模組中), 1934
internal_attr (*zipfile.ZipInfo* 的屬性), 600
Internaldate2tuple() (於 *imaplib* 模組中), 1484
InternalError, 552
internalSubset (*xml.dom.DocumentType* 的屬性), 1388
INTERNET_TIMEOUT (於 *test.support* 模組中), 1837
Internet（網際網路）, 1423

interpolated string literal（插值字串常數）, 63
Interpolation (*ast* 中的類🄵), 2099
Interpolation (*string.templatelib* 中的類🄵), 137
InterpolationDepthError, 643
InterpolationError, 643
InterpolationMissingOptionError, 643
interpolations (*string.templatelib.Template* 的屬性), 135
InterpolationSyntaxError, 643
interpolation（插值）
    bytearray (%), 80
    bytes (%), 80
interpolation（插值）、字串 (%), 64
interpreted（直譯的）, **2235**
Interpreter (*concurrent.interpreters* 中的類🄵), 1042
interpreter prompts（直譯器提示）, 1939
interpreter shutdown（直譯器關閉）, **2235**
interpreter_requires_environment() (於 *test.support.script_helper* 模組中), 1846
InterpreterError, 1042
InterpreterNotFoundError, 1042
InterpreterPoolExecutor (*concurrent.futures* 中的類🄵), 1034
interrupt() (*multiprocessing.Process* 的方法), 991
interrupt() (*sqlite3.Connection* 的方法), 542
interrupt_main() (於 *_thread* 模組中), 1072
InterruptedError, 117
intersection() (*frozenset* 的方法), 90
intersection_update() (*frozenset* 的方法), 91
IntFlag (*enum* 中的類🄵), 329
intro (*cmd.Cmd* 的屬性), 961
InuseAttributeErr, 1392
inv() (於 *operator* 模組中), 436
inv_cdf() (*statistics.NormalDist* 的方法), 404
InvalidAccessErr, 1392
invalidate_caches() (*importlib.abc.MetaPathFinder* 的方法), 2062
invalidate_caches() (*importlib.abc.PathEntryFinder* 的方法), 2062
invalidate_caches() (*importlib.machinery.FileFinder* 的方法), 2070
invalidate_caches() (*importlib.machinery.PathFinder* 的類🄵方法), 2069
invalidate_caches() (於 *importlib* 模組中), 2060
invalidate_caches() (於 *platform* 模組中), 806
invalidate_caches() (*zipimport.zipimporter* 的方法), 2052
--invalidation-mode
    compileall 命令列選項, 2148
InvalidBase64CharactersDefect, 1286
InvalidBase64LengthDefect, 1286
InvalidBase64PaddingDefect, 1286
InvalidCharacterErr, 1392
InvalidDateDefect, 1286
InvalidFileException, 649
InvalidModificationErr, 1392
InvalidOperation (*decimal* 中的類🄵), 372
InvalidStateErr, 1392
InvalidStateError, 1039, 1118
InvalidTZPathWarning, 255
InvalidURL, 1468
InvalidWriteError, 644
Invert (*ast* 中的類🄵), 2102
invert() (於 *operator* 模組中), 436
io
    module, 734
IO (*typing* 中的類🄵), 1705
IO_REPARSE_TAG_APPEXECLINK (於 *stat* 模組中), 480
IO_REPARSE_TAG_MOUNT_POINT (於 *stat* 模組中), 480
IO_REPARSE_TAG_SYMLINK (於 *stat* 模組中), 480
IOBase (*io* 中的類🄵), 737
ioctl() (*socket.socket* 的方法), 1193
ioctl() (於 *fcntl* 模組中), 2204
IOCTL_VM_SOCKETS_GET_LOCAL_CID (於 *socket* 模組中), 1181
IOError, 116
ior() (於 *operator* 模組中), 441
ios_ver() (於 *platform* 模組中), 807
io.StringIO
    object（物件）, 53
ip (*ipaddress.IPv4Interface* 的屬性), 1554
ip (*ipaddress.IPv6Interface* 的屬性), 1555
ip_address() (於 *ipaddress* 模組中), 1543
ip_interface() (於 *ipaddress* 模組中), 1544
ip_network() (於 *ipaddress* 模組中), 1543
ipaddress
    module, 1543
ipow() (於 *operator* 模組中), 441
ipv4_mapped (*ipaddress.IPv6Address* 的屬性), 1547
IPv4Address (*ipaddress* 中的類🄵), 1544
IPv4Interface (*ipaddress* 中的類🄵), 1554
IPv4Network (*ipaddress* 中的類🄵), 1549
IPV6_ENABLED (於 *test.support.socket_helper* 模組中), 1846
ipv6_mapped (*ipaddress.IPv4Address* 的屬性), 1546
IPv6Address (*ipaddress* 中的類🄵), 1546
IPv6Interface (*ipaddress* 中的類🄵), 1555
IPv6Network (*ipaddress* 中的類🄵), 1552
irshift() (於 *operator* 模組中), 441
is
    operator（運算子）, 38
Is (*ast* 中的類🄵), 2103
is not
    operator（運算子）, 38
is_() (於 *operator* 模組中), 436
is_absolute() (*pathlib.PurePath* 的方法), 450
is_active() (*graphlib.TopologicalSorter* 的方法), 335
is_alive() (*multiprocessing.Process* 的方法), 990
is_alive() (*threading.Thread* 的方法), 973
is_android (於 *test.support* 模組中), 1837

is_annotated() (*symtable.Symbol* 的方法), 2132
is_anonymous (*ctypes.CField* 的屬性), 852
is_apple (於 *test.support* 模組中), 1837
is_apple_mobile (於 *test.support* 模組中), 1837
is_assigned() (*symtable.Symbol* 的方法), 2133
is_async (*pyclbr.Function* 的屬性), 2144
is_attachment() (*email.message.EmailMessage* 的方法), 1269
is_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 的方法), 1446
is_bitfield (*ctypes.CField* 的屬性), 852
is_block_device() (*pathlib.Path* 的方法), 458
is_blocked() (*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
is_canonical() (*decimal.Context* 的方法), 368
is_canonical() (*decimal.Decimal* 的方法), 360
is_char_device() (*pathlib.Path* 的方法), 458
IS_CHARACTER_JUNK() (於 *difflib* 模組中), 165
is_check_supported() (於 *lzma* 模組中), 589
is_closed() (*asyncio.loop* 的方法), 1122
is_closing() (*asyncio.BaseTransport* 的方法), 1148
is_closing() (*asyncio.StreamWriter* 的方法), 1102
is_comp_cell() (*symtable.Symbol* 的方法), 2133
is_comp_iter() (*symtable.Symbol* 的方法), 2133
is_dataclass() (於 *dataclasses* 模組中), 1975
is_declared_global() (*symtable.Symbol* 的方法), 2132
is_dir() (*importlib.abc.Traversable* 的方法), 2067
is_dir() (*importlib.resources.abc.Traversable* 的方法), 2083
is_dir() (*os.DirEntry* 的方法), 701
is_dir() (*pathlib.Path* 的方法), 457
is_dir() (*pathlib.types.PathInfo* 的方法), 468
is_dir() (*zipfile.Path* 的方法), 597
is_dir() (*zipfile.ZipInfo* 的方法), 599
is_emscripten (於 *test.support* 模組中), 1837
is_enabled() (於 *faulthandler* 模組中), 1866
is_expired() (*http.cookiejar.Cookie* 的方法), 1529
is_fifo() (*pathlib.Path* 的方法), 458
is_file() (*importlib.abc.Traversable* 的方法), 2067
is_file() (*importlib.resources.abc.Traversable* 的方法), 2083
is_file() (*os.DirEntry* 的方法), 701
is_file() (*pathlib.Path* 的方法), 457
is_file() (*pathlib.types.PathInfo* 的方法), 468
is_file() (*zipfile.Path* 的方法), 597
is_finalized() (於 *gc* 模組中), 2014
is_finalizing() (於 *sys* 模組中), 1935
is_finite() (*decimal.Context* 的方法), 368
is_finite() (*decimal.Decimal* 的方法), 361
is_free() (*symtable.Symbol* 的方法), 2132
is_free_class() (*symtable.Symbol* 的方法), 2132
is_global (*ipaddress.IPv4Address* 的屬性), 1545
is_global (*ipaddress.IPv6Address* 的屬性), 1547
is_global() (*symtable.Symbol* 的方法), 2132
is_hop_by_hop() (於 *wsgiref.util* 模組中), 1427
is_imported() (*symtable.Symbol* 的方法), 2132
is_infinite() (*decimal.Context* 的方法), 368
is_infinite() (*decimal.Decimal* 的方法), 361
is_integer() (*float* 的方法), 43
is_integer() (*fractions.Fraction* 的方法), 382
is_integer() (*int* 的方法), 42
is_junction() (*os.DirEntry* 的方法), 701
is_junction() (*pathlib.Path* 的方法), 458
is_jython (於 *test.support* 模組中), 1837
IS_LINE_JUNK() (於 *difflib* 模組中), 165
is_linetouched() (*curses.window* 的方法), 940
is_link_local (*ipaddress.IPv4Address* 的屬性), 1546
is_link_local (*ipaddress.IPv4Network* 的屬性), 1550
is_link_local (*ipaddress.IPv6Address* 的屬性), 1547
is_link_local (*ipaddress.IPv6Network* 的屬性), 1553
is_local() (*symtable.Symbol* 的方法), 2132
is_loopback (*ipaddress.IPv4Address* 的屬性), 1546
is_loopback (*ipaddress.IPv4Network* 的屬性), 1550
is_loopback (*ipaddress.IPv6Address* 的屬性), 1547
is_loopback (*ipaddress.IPv6Network* 的屬性), 1553
is_mount() (*pathlib.Path* 的方法), 458
is_multicast (*ipaddress.IPv4Address* 的屬性), 1545
is_multicast (*ipaddress.IPv4Network* 的屬性), 1550
is_multicast (*ipaddress.IPv6Address* 的屬性), 1547
is_multicast (*ipaddress.IPv6Network* 的屬性), 1552
is_multipart() (*email.message.EmailMessage* 的方法), 1265
is_multipart() (*email.message.Message* 的方法), 1302
is_namespace() (*symtable.Symbol* 的方法), 2133
is_nan() (*decimal.Context* 的方法), 368
is_nan() (*decimal.Decimal* 的方法), 361
is_nested() (*symtable.SymbolTable* 的方法), 2131
is_none() (於 *operator* 模組中), 436
is_nonlocal() (*symtable.Symbol* 的方法), 2132
is_normal() (*decimal.Context* 的方法), 368
is_normal() (*decimal.Decimal* 的方法), 361
is_normalized() (於 *unicodedata* 模組中), 178
is_not() (於 *operator* 模組中), 436
is_not_allowed() (*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
is_not_none() (於 *operator* 模組中), 436
IS_OP (*opcode*), 2165
is_optimized() (*symtable.SymbolTable* 的方法), 2131
is_package() (*importlib.abc.InspectLoader* 的方法), 2064
is_package() (*importlib.abc.SourceLoader* 的方法), 2066
is_package() (*importlib.machinery.ExtensionFileLoader* 的方法), 2071
is_package() (*importlib.machinery.SourceFileLoader* 的方法), 2070
is_package() (*importlib.machinery.SourcelessFileLoader* 的方法), 2071

is_package() (*zipimport.zipimporter* 的方法), 2052
is_parameter() (*symtable.Symbol* 的方法), 2132
is_private (*ipaddress.IPv4Address* 的屬性), 1545
is_private (*ipaddress.IPv4Network* 的屬性), 1550
is_private (*ipaddress.IPv6Address* 的屬性), 1547
is_private (*ipaddress.IPv6Network* 的屬性), 1552
is_protocol() (於 *typing* 模組中), 1712
is_python_build() (於 *sysconfig* 模組中), 1956
is_qnan() (*decimal.Context* 的方法), 368
is_qnan() (*decimal.Decimal* 的方法), 361
is_reading() (*asyncio.ReadTransport* 的方法), 1149
is_referenced() (*symtable.Symbol* 的方法), 2132
is_relative_to() (*pathlib.PurePath* 的方法), 450
is_reserved (*ipaddress.IPv4Address* 的屬性), 1546
is_reserved (*ipaddress.IPv4Network* 的屬性), 1550
is_reserved (*ipaddress.IPv6Address* 的屬性), 1547
is_reserved (*ipaddress.IPv6Network* 的屬性), 1553
is_reserved() (*pathlib.PurePath* 的方法), 451
is_resource() (*importlib.abc.ResourceReader* 的方法), 2067
is_resource() (*importlib.resources.abc.ResourceReader* 的方法), 2082
is_resource() (於 *importlib.resources* 模組中), 2081
is_resource_enabled() (於 *test.support* 模組中), 1839
is_running() (*asyncio.loop* 的方法), 1122
is_running() (*concurrent.interpreters.Interpreter* 的方法), 1042
is_safe (*uuid.UUID* 的屬性), 1499
is_serving() (*asyncio.Server* 的方法), 1140
is_set() (*asyncio.Event* 的方法), 1107
is_set() (*threading.Event* 的方法), 979
is_signed() (*decimal.Context* 的方法), 368
is_signed() (*decimal.Decimal* 的方法), 361
is_site_local (*ipaddress.IPv6Address* 的屬性), 1547
is_site_local (*ipaddress.IPv6Network* 的屬性), 1553
is_skipped_line() (*bdb.Bdb* 的方法), 1862
is_snan() (*decimal.Context* 的方法), 369
is_snan() (*decimal.Decimal* 的方法), 361
is_socket() (*pathlib.Path* 的方法), 458
is_stack_trampoline_active() (於 *sys* 模組中), 1942
is_subnormal() (*decimal.Context* 的方法), 369
is_subnormal() (*decimal.Decimal* 的方法), 361
is_symlink() (*os.DirEntry* 的方法), 701
is_symlink() (*pathlib.Path* 的方法), 458
is_symlink() (*pathlib.types.PathInfo* 的方法), 468
is_symlink() (*zipfile.Path* 的方法), 597
is_tarfile() (於 *tarfile* 模組中), 604
is_term_resized() (於 *curses* 模組中), 932
is_tracing() (於 *tracemalloc* 模組中), 1898
is_tracked() (於 *gc* 模組中), 2014
is_type_parameter() (*symtable.Symbol* 的方法), 2132
is_typeddict() (於 *typing* 模組中), 1712
is_unspecified (*ipaddress.IPv4Address* 的屬性), 1546
is_unspecified (*ipaddress.IPv4Network* 的屬性), 1550
is_unspecified (*ipaddress.IPv6Address* 的屬性), 1547
is_unspecified (*ipaddress.IPv6Network* 的屬性), 1553
is_valid() (*string.Template* 的方法), 133
is_wasi (於 *test.support* 模組中), 1837
is_wintouched() (*curses.window* 的方法), 940
is_zero() (*decimal.Context* 的方法), 369
is_zero() (*decimal.Decimal* 的方法), 361
is_zipfile() (於 *zipfile* 模組中), 592
isabs() (於 *os.path* 模組中), 471
isabstract() (於 *inspect* 模組中), 2020
IsADirectoryError, 117
isalnum() (*bytearray* 的方法), 76
isalnum() (*bytes* 的方法), 76
isalnum() (*str* 的方法), 57
isalnum() (於 *curses.ascii* 模組中), 957
isalpha() (*bytearray* 的方法), 76
isalpha() (*bytes* 的方法), 76
isalpha() (*str* 的方法), 57
isalpha() (於 *curses.ascii* 模組中), 957
isascii() (*bytearray* 的方法), 77
isascii() (*bytes* 的方法), 77
isascii() (*str* 的方法), 57
isascii() (於 *curses.ascii* 模組中), 957
isasyncgen() (於 *inspect* 模組中), 2020
isasyncgenfunction() (於 *inspect* 模組中), 2020
isatty() (*io.IOBase* 的方法), 738
isatty() (於 *os* 模組中), 680
isawaitable() (於 *inspect* 模組中), 2019
isblank() (於 *curses.ascii* 模組中), 957
isblk() (*tarfile.TarInfo* 的方法), 612
isbuiltin() (於 *inspect* 模組中), 2020
ischr() (*tarfile.TarInfo* 的方法), 612
isclass() (於 *inspect* 模組中), 2018
isclose() (於 *cmath* 模組中), 352
isclose() (於 *math* 模組中), 343
iscntrl() (於 *curses.ascii* 模組中), 957
iscode() (於 *inspect* 模組中), 2020
iscoroutine() (於 *asyncio* 模組中), 1094
iscoroutine() (於 *inspect* 模組中), 2019
iscoroutinefunction() (於 *inspect* 模組中), 2019
isctrl() (於 *curses.ascii* 模組中), 957
isDaemon() (*threading.Thread* 的方法), 973
isdatadescriptor() (於 *inspect* 模組中), 2021
isdecimal() (*str* 的方法), 57
isdev() (*tarfile.TarInfo* 的方法), 612
isdevdrive() (於 *os.path* 模組中), 471
isdigit() (*bytearray* 的方法), 77
isdigit() (*bytes* 的方法), 77
isdigit() (*str* 的方法), 57
isdigit() (於 *curses.ascii* 模組中), 957
isdir() (*tarfile.TarInfo* 的方法), 612
isdir() (於 *os.path* 模組中), 471

isdisjoint() (*frozenset* 的方法), 90
isdown() (於 *turtle* 模組中), 1643
iselement() (於 *xml.etree.ElementTree* 模組中), 1374
isenabled() (於 *gc* 模組中), 2012
isEnabledFor() (*logging.Logger* 的方法), 762
isendwin() (於 *curses* 模組中), 932
ISEOF() (於 *token* 模組中), 2134
isfifo() (*tarfile.TarInfo* 的方法), 612
isfile() (*tarfile.TarInfo* 的方法), 611
isfile() (於 *os.path* 模組中), 471
isfinite() (於 *cmath* 模組中), 352
isfinite() (於 *math* 模組中), 344
isfirstline() (於 *fileinput* 模組中), 927
isframe() (於 *inspect* 模組中), 2020
isfunction() (於 *inspect* 模組中), 2019
isfuture() (於 *asyncio* 模組中), 1144
isgenerator() (於 *inspect* 模組中), 2019
isgeneratorfunction() (於 *inspect* 模組中), 2019
isgetsetdescriptor() (於 *inspect* 模組中), 2021
isgraph() (於 *curses.ascii* 模組中), 957
isidentifier() (*str* 的方法), 57
isinf() (於 *cmath* 模組中), 352
isinf() (於 *math* 模組中), 344
isinstance()
    built-in function, 19
isjunction() (於 *os.path* 模組中), 471
iskeyword() (於 *keyword* 模組中), 2139
isleap() (於 *calendar* 模組中), 259
islice() (於 *itertools* 模組中), 415
islink() (於 *os.path* 模組中), 471
islnk() (*tarfile.TarInfo* 的方法), 612
islower() (*bytearray* 的方法), 77
islower() (*bytes* 的方法), 77
islower() (*str* 的方法), 57
islower() (於 *curses.ascii* 模組中), 957
ismemberdescriptor() (於 *inspect* 模組中), 2021
ismeta() (於 *curses.ascii* 模組中), 957
ismethod() (於 *inspect* 模組中), 2018
ismethoddescriptor() (於 *inspect* 模組中), 2020
ismethodwrapper() (於 *inspect* 模組中), 2020
ismodule() (於 *inspect* 模組中), 2018
ismount() (於 *os.path* 模組中), 471
isnan() (於 *cmath* 模組中), 352
isnan() (於 *math* 模組中), 344
ISNONTERMINAL() (於 *token* 模組中), 2134
IsNot (*ast* 中的類🄵), 2103
isnumeric() (*str* 的方法), 57
isocalendar() (*datetime.date* 的方法), 222
isocalendar() (*datetime.datetime* 的方法), 232
isoformat() (*datetime.date* 的方法), 223
isoformat() (*datetime.datetime* 的方法), 232
isoformat() (*datetime.time* 的方法), 238
isolated (*sys.flags* 的屬性), 1926
IsolatedAsyncioTestCase (*unittest* 中的類🄵), 1765
isolation_level (*sqlite3.Connection* 的屬性), 547
isoweekday() (*datetime.date* 的方法), 222
isoweekday() (*datetime.datetime* 的方法), 232
ispackage() (於 *inspect* 模組中), 2019
isprint() (於 *curses.ascii* 模組中), 957
isprintable() (*str* 的方法), 57
ispunct() (於 *curses.ascii* 模組中), 957
isqrt() (於 *math* 模組中), 342
isreadable() (*pprint.PrettyPrinter* 的方法), 313
isreadable() (於 *pprint* 模組中), 312
isrecursive() (*pprint.PrettyPrinter* 的方法), 313
isrecursive() (於 *pprint* 模組中), 312
isreg() (*tarfile.TarInfo* 的方法), 612
isreserved() (於 *os.path* 模組中), 471
isReservedKey() (*http.cookies.Morsel* 的方法), 1520
isroutine() (於 *inspect* 模組中), 2020
isSameNode() (*xml.dom.Node* 的方法), 1387
issoftkeyword() (於 *keyword* 模組中), 2139
isspace() (*bytearray* 的方法), 77
isspace() (*bytes* 的方法), 77
isspace() (*str* 的方法), 58
isspace() (於 *curses.ascii* 模組中), 957
isstdin() (於 *fileinput* 模組中), 927
issubclass()
    built-in function, 19
issubset() (*frozenset* 的方法), 90
issuperset() (*frozenset* 的方法), 90
issym() (*tarfile.TarInfo* 的方法), 612
ISTERMINAL() (於 *token* 模組中), 2134
istitle() (*bytearray* 的方法), 77
istitle() (*bytes* 的方法), 77
istitle() (*str* 的方法), 58
istraceback() (於 *inspect* 模組中), 2020
isub() (於 *operator* 模組中), 442
isupper() (*bytearray* 的方法), 77
isupper() (*bytes* 的方法), 77
isupper() (*str* 的方法), 58
isupper() (於 *curses.ascii* 模組中), 957
isvisible() (於 *turtle* 模組中), 1647
isxdigit() (於 *curses.ascii* 模組中), 957
ITALIC (於 *tkinter.font* 模組中), 1591
item() (*tkinter.ttk.Treeview* 的方法), 1611
item() (*xml.dom.NamedNodeMap* 的方法), 1391
item() (*xml.dom.NodeList* 的方法), 1388
itemgetter() (於 *operator* 模組中), 439
items() (*configparser.ConfigParser* 的方法), 641
items() (*contextvars.Context* 的方法), 1070
items() (*dict* 的方法), 94
items() (*email.message.EmailMessage* 的方法), 1266
items() (*email.message.Message* 的方法), 1304
items() (*mailbox.Mailbox* 的方法), 1331
items() (*types.MappingProxyType* 的方法), 308
items() (*xml.etree.ElementTree.Element* 的方法), 1377
itemsize (*array.array* 的屬性), 294
itemsize (*memoryview* 的屬性), 88
ItemsView (*collections.abc* 中的類🄵), 284
ItemsView (*typing* 中的類🄵), 1717
iter()
    built-in function, 19
iter() (*xml.etree.ElementTree.Element* 的方法), 1378

iter() (*xml.etree.ElementTree.ElementTree* 的方法), 1379
iter_attachments() (*email.message.EmailMessage* 的方法), 1270
iter_child_nodes() (於 *ast* 模組中), 2126
iter_fields() (於 *ast* 模組中), 2126
iter_importers() (於 *pkgutil* 模組中), 2054
iter_modules() (於 *pkgutil* 模組中), 2054
iter_parts() (*email.message.EmailMessage* 的方法), 1270
iter_unpack() (*struct.Struct* 的方法), 193
iter_unpack() (於 *struct* 模組中), 188
Iterable (*collections.abc* 中的類🄵), 283
Iterable (*typing* 中的類🄵), 1718
iterable（可🄵代物件）, **2235**
Iterator (*collections.abc* 中的類🄵), 283
Iterator (*typing* 中的類🄵), 1718
iterator protocol（🄵代器協定）, 45
iterator（🄵代器）, **2235**
iterdecode() (於 *codecs* 模組中), 196
iterdir() (*importlib.abc.Traversable* 的方法), 2067
iterdir() (*importlib.resources.abc.Traversable* 的方法), 2082
iterdir() (*pathlib.Path* 的方法), 460
iterdir() (*zipfile.Path* 的方法), 597
iterdump() (*sqlite3.Connection* 的方法), 544
iterencode() (*json.JSONEncoder* 的方法), 1326
iterencode() (於 *codecs* 模組中), 196
iterfind() (*xml.etree.ElementTree.Element* 的方法), 1378
iterfind() (*xml.etree.ElementTree.ElementTree* 的方法), 1380
iteritems() (*mailbox.Mailbox* 的方法), 1331
iterkeys() (*mailbox.Mailbox* 的方法), 1330
itermonthdates() (*calendar.Calendar* 的方法), 256
itermonthdays() (*calendar.Calendar* 的方法), 256
itermonthdays2() (*calendar.Calendar* 的方法), 256
itermonthdays3() (*calendar.Calendar* 的方法), 256
itermonthdays4() (*calendar.Calendar* 的方法), 256
iterparse() (於 *xml.etree.ElementTree* 模組中), 1374
itertext() (*xml.etree.ElementTree.Element* 的方法), 1378
itertools
    module, 409
itervalues() (*mailbox.Mailbox* 的方法), 1330
iterweekdays() (*calendar.Calendar* 的方法), 256
ITIMER_PROF (於 *signal* 模組中), 1250
ITIMER_REAL (於 *signal* 模組中), 1250
ITIMER_VIRTUAL (於 *signal* 模組中), 1250
ItimerError, 1250
itruediv() (於 *operator* 模組中), 442
ixor() (於 *operator* 模組中), 442

J

-j
    compileall 命令列選項, 2148
JANUARY (於 *calendar* 模組中), 261
java_ver() (於 *platform* 模組中), 806
job_size (*compression.zstd.CompressionParameter* 的屬性), 571
join() (*asyncio.Queue* 的方法), 1116
join() (*bytearray* 的方法), 71
join() (*bytes* 的方法), 71
join() (*multiprocessing.JoinableQueue* 的方法), 995
join() (*multiprocessing.pool.Pool* 的方法), 1012
join() (*multiprocessing.Process* 的方法), 990
join() (*queue.Queue* 的方法), 1066
join() (*str* 的方法), 58
join() (*threading.Thread* 的方法), 972
join() (於 *os.path* 模組中), 472
join() (於 *shlex* 模組中), 2191
join_thread() (*multiprocessing.Queue* 的方法), 994
join_thread() (於 *test.support.threading_helper* 模組中), 1848
JoinableQueue (*multiprocessing* 中的類🄵), 995
JoinedStr (*ast* 中的類🄵), 2099
joinpath() (*importlib.abc.Traversable* 的方法), 2067
joinpath() (*importlib.resources.abc.Traversable* 的方法), 2083
joinpath() (*pathlib.PurePath* 的方法), 451
joinpath() (*zipfile.Path* 的方法), 598
js_output() (*http.cookies.BaseCookie* 的方法), 1519
js_output() (*http.cookies.Morsel* 的方法), 1520
json
    module, 1319
json 命令列選項
    --compact, 1329
    -h, 1329
    --help, 1329
    --indent, 1329
    infile, 1328
    --json-lines, 1329
    --no-ensure-ascii, 1329
    --no-indent, 1329
    outfile, 1328
    --sort-keys, 1328
    --tab, 1329
JSONDecodeError, 1326
JSONDecoder (*json* 中的類🄵), 1324
JSONEncoder (*json* 中的類🄵), 1325
--json-lines
    json 命令列選項, 1329
json.tool
    module, 1328
JULY (於 *calendar* 模組中), 261
JUMP (*monitoring event*), 1948
JUMP (*opcode*), 2172
jump (*pdb command*), 1874
JUMP_BACKWARD (*opcode*), 2166
JUMP_BACKWARD_NO_INTERRUPT (*opcode*), 2166
JUMP_FORWARD (*opcode*), 2166
JUMP_IF_FALSE (*opcode*), 2172
JUMP_IF_TRUE (*opcode*), 2172
JUMP_NO_INTERRUPT (*opcode*), 2172
JUNE (於 *calendar* 模組中), 261

K

-k
    unittest 命令列選項, 1749
kbhit() (於 *msvcrt* 模組中), 2178
kde() (於 *statistics* 模組中), 396
kde_random() (於 *statistics* 模組中), 397
KEEP (*enum.FlagBoundary* 的屬性), 331
kevent() (於 *select* 模組中), 1238
key (*http.cookies.Morsel* 的屬性), 1520
key (*zoneinfo.ZoneInfo* 的屬性), 253
key function（鍵函式）, **2236**
KEY_A1 (於 *curses* 模組中), 945
KEY_A3 (於 *curses* 模組中), 946
KEY_ALL_ACCESS (於 *winreg* 模組中), 2186
KEY_B2 (於 *curses* 模組中), 946
KEY_BACKSPACE (於 *curses* 模組中), 944
KEY_BEG (於 *curses* 模組中), 946
KEY_BREAK (於 *curses* 模組中), 944
KEY_BTAB (於 *curses* 模組中), 946
KEY_C1 (於 *curses* 模組中), 946
KEY_C3 (於 *curses* 模組中), 946
KEY_CANCEL (於 *curses* 模組中), 946
KEY_CATAB (於 *curses* 模組中), 945
KEY_CLEAR (於 *curses* 模組中), 945
KEY_CLOSE (於 *curses* 模組中), 946
KEY_COMMAND (於 *curses* 模組中), 946
KEY_COPY (於 *curses* 模組中), 946
KEY_CREATE (於 *curses* 模組中), 946
KEY_CREATE_LINK (於 *winreg* 模組中), 2186
KEY_CREATE_SUB_KEY (於 *winreg* 模組中), 2186
KEY_CTAB (於 *curses* 模組中), 945
KEY_DC (於 *curses* 模組中), 944
KEY_DL (於 *curses* 模組中), 944
KEY_DOWN (於 *curses* 模組中), 944
KEY_EIC (於 *curses* 模組中), 945
KEY_END (於 *curses* 模組中), 946
KEY_ENTER (於 *curses* 模組中), 945
KEY_ENUMERATE_SUB_KEYS (於 *winreg* 模組中), 2186
KEY_EOL (於 *curses* 模組中), 945
KEY_EOS (於 *curses* 模組中), 945
KEY_EXECUTE (於 *winreg* 模組中), 2186
KEY_EXIT (於 *curses* 模組中), 946
KEY_F0 (於 *curses* 模組中), 944
KEY_FIND (於 *curses* 模組中), 946
KEY_Fn (於 *curses* 模組中), 944
KEY_HELP (於 *curses* 模組中), 946
KEY_HOME (於 *curses* 模組中), 944
KEY_IC (於 *curses* 模組中), 945
KEY_IL (於 *curses* 模組中), 944
KEY_LEFT (於 *curses* 模組中), 944
KEY_LL (於 *curses* 模組中), 945
KEY_MARK (於 *curses* 模組中), 946
KEY_MAX (於 *curses* 模組中), 949
KEY_MESSAGE (於 *curses* 模組中), 946
KEY_MIN (於 *curses* 模組中), 944
KEY_MOUSE (於 *curses* 模組中), 949
KEY_MOVE (於 *curses* 模組中), 946
KEY_NEXT (於 *curses* 模組中), 947
KEY_NOTIFY (於 *winreg* 模組中), 2186
KEY_NPAGE (於 *curses* 模組中), 945
KEY_OPEN (於 *curses* 模組中), 947
KEY_OPTIONS (於 *curses* 模組中), 947
KEY_PPAGE (於 *curses* 模組中), 945
KEY_PREVIOUS (於 *curses* 模組中), 947
KEY_PRINT (於 *curses* 模組中), 945
KEY_QUERY_VALUE (於 *winreg* 模組中), 2186
KEY_READ (於 *winreg* 模組中), 2186
KEY_REDO (於 *curses* 模組中), 947
KEY_REFERENCE (於 *curses* 模組中), 947
KEY_REFRESH (於 *curses* 模組中), 947
KEY_REPLACE (於 *curses* 模組中), 947
KEY_RESET (於 *curses* 模組中), 945
KEY_RESIZE (於 *curses* 模組中), 949
KEY_RESTART (於 *curses* 模組中), 947
KEY_RESUME (於 *curses* 模組中), 947
KEY_RIGHT (於 *curses* 模組中), 944
KEY_SAVE (於 *curses* 模組中), 947
KEY_SBEG (於 *curses* 模組中), 947
KEY_SCANCEL (於 *curses* 模組中), 947
KEY_SCOMMAND (於 *curses* 模組中), 947
KEY_SCOPY (於 *curses* 模組中), 947
KEY_SCREATE (於 *curses* 模組中), 947
KEY_SDC (於 *curses* 模組中), 947
KEY_SDL (於 *curses* 模組中), 947
KEY_SELECT (於 *curses* 模組中), 948
KEY_SEND (於 *curses* 模組中), 948
KEY_SEOL (於 *curses* 模組中), 948
KEY_SET_VALUE (於 *winreg* 模組中), 2186
KEY_SEXIT (於 *curses* 模組中), 948
KEY_SF (於 *curses* 模組中), 945
KEY_SFIND (於 *curses* 模組中), 948
KEY_SHELP (於 *curses* 模組中), 948
KEY_SHOME (於 *curses* 模組中), 948
KEY_SIC (於 *curses* 模組中), 948
KEY_SLEFT (於 *curses* 模組中), 948
KEY_SMESSAGE (於 *curses* 模組中), 948
KEY_SMOVE (於 *curses* 模組中), 948
KEY_SNEXT (於 *curses* 模組中), 948
KEY_SOPTIONS (於 *curses* 模組中), 948
KEY_SPREVIOUS (於 *curses* 模組中), 948
KEY_SPRINT (於 *curses* 模組中), 948
KEY_SR (於 *curses* 模組中), 945
KEY_SREDO (於 *curses* 模組中), 948
KEY_SREPLACE (於 *curses* 模組中), 948
KEY_SRESET (於 *curses* 模組中), 945
KEY_SRIGHT (於 *curses* 模組中), 948
KEY_SRSUME (於 *curses* 模組中), 949
KEY_SSAVE (於 *curses* 模組中), 949
KEY_SSUSPEND (於 *curses* 模組中), 949
KEY_STAB (於 *curses* 模組中), 945
KEY_SUNDO (於 *curses* 模組中), 949
KEY_SUSPEND (於 *curses* 模組中), 949
KEY_UNDO (於 *curses* 模組中), 949
KEY_UP (於 *curses* 模組中), 944
KEY_WOW64_32KEY (於 *winreg* 模組中), 2186
KEY_WOW64_64KEY (於 *winreg* 模組中), 2186

KEY_WRITE (於 *winreg* 模組中), 2186
KeyboardInterrupt, 112
KeyError, 111
keylog_filename (*ssl.SSLContext* 的屬性), 1224
keyname() (於 *curses* 模組中), 932
keypad() (*curses.window* 的方法), 940
keyrefs() (*weakref.WeakKeyDictionary* 的方法), 298
keys() (*contextvars.Context* 的方法), 1070
keys() (*dict* 的方法), 94
keys() (*email.message.EmailMessage* 的方法), 1266
keys() (*email.message.Message* 的方法), 1304
keys() (*mailbox.Mailbox* 的方法), 1330
keys() (*sqlite3.Row* 的方法), 550
keys() (*types.MappingProxyType* 的方法), 308
keys() (*xml.etree.ElementTree.Element* 的方法), 1377
KeysView (*collections.abc* 中的類🄵), 284
KeysView (*typing* 中的類🄵), 1717
keyword
    module, 2139
keyword (*ast* 中的類🄵), 2103
keyword argument（關鍵字引數）, **2236**
keywords (*functools.partial* 的屬性), 435
kill() (*asyncio.subprocess.Process* 的方法), 1114
kill() (*asyncio.SubprocessTransport* 的方法), 1151
kill() (*multiprocessing.Process* 的方法), 991
kill() (*subprocess.Popen* 的方法), 1054
kill() (於 *os* 模組中), 720
kill_python() (於 *test.support.script_helper* 模組中), 1847
kill_workers() (*concurrent.futures.ProcessPoolExecutor* 的方法), 1036
killchar() (於 *curses* 模組中), 932
killpg() (於 *os* 模組中), 720
kind (*inspect.Parameter* 的屬性), 2024
knownfiles (於 *mimetypes* 模組中), 1349
kqueue() (於 *select* 模組中), 1238
KqueueSelector (*selectors* 中的類🄵), 1246
KW_ONLY (於 *dataclasses* 模組中), 1976
kwargs (*inspect.BoundArguments* 的屬性), 2026
kwargs (*typing.ParamSpec* 的屬性), 1696
kwlist (於 *keyword* 模組中), 2139

L

-L
    calendar 命令列選項, 262
-l
    calendar 命令列選項, 263
    compileall 命令列選項, 2147
    mimetypes 命令列選項, 1351
    pickletools 命令列選項, 2174
    tarfile 命令列選項, 616
    trace 命令列選項, 1891
    zipfile 命令列選項, 601
L (於 *re* 模組中), 147
L2CAP_LM (於 *socket* 模組中), 1182
lambda, **2236**
Lambda (*ast* 中的類🄵), 2121
LambdaType (於 *types* 模組中), 305
LANG, 1561, 1562, 1569, 1573
LANGUAGE, 1561, 1562
language（語言）
    C, 38
large files（大型檔案）, 2196
LARGEST (於 *test.support* 模組中), 1838
LargeZipFile, 591
last_accepted (*multiprocessing.connection.Listener* 的屬性), 1014
last_exc (於 *sys* 模組中), 1935
last_mode (*compression.zstd.ZstdCompressor* 的屬性), 566
last_traceback (於 *sys* 模組中), 1936
last_type (於 *sys* 模組中), 1936
last_value (於 *sys* 模組中), 1936
lastChild (*xml.dom.Node* 的屬性), 1387
lastcmd (*cmd.Cmd* 的屬性), 961
lastgroup (*re.Match* 的屬性), 155
lastindex (*re.Match* 的屬性), 155
lastResort (於 *logging* 模組中), 777
lastrowid (*sqlite3.Cursor* 的屬性), 550
layout() (*tkinter.ttk.Style* 的方法), 1613
lazy (*compression.zstd.Strategy* 的屬性), 572
lazy2 (*compression.zstd.Strategy* 的屬性), 572
lazycache() (於 *linecache* 模組中), 492
LazyLoader (*importlib.util* 中的類🄵), 2075
LBRACE (於 *token* 模組中), 2137
LBYL, **2236**
LC_ALL, 1561, 1562
LC_ALL (於 *locale* 模組中), 1575
LC_COLLATE (於 *locale* 模組中), 1575
LC_CTYPE (於 *locale* 模組中), 1575
LC_MESSAGES, 1561, 1562
LC_MESSAGES (於 *locale* 模組中), 1575
LC_MONETARY (於 *locale* 模組中), 1575
LC_NUMERIC (於 *locale* 模組中), 1575
LC_TIME (於 *locale* 模組中), 1575
lchflags() (於 *os* 模組中), 693
lchmod() (*pathlib.Path* 的方法), 465
lchmod() (於 *os* 模組中), 694
lchown() (於 *os* 模組中), 694
lcm() (於 *math* 模組中), 342
ldexp() (於 *math* 模組中), 344
ldm_bucket_size_log (*compression.zstd.CompressionParameter* 的屬性), 571
ldm_hash_log (*compression.zstd.CompressionParameter* 的屬性), 570
ldm_hash_rate_log (*compression.zstd.CompressionParameter* 的屬性), 571
ldm_min_match (*compression.zstd.CompressionParameter* 的屬性), 570
le() (於 *operator* 模組中), 435
leapdays() (於 *calendar* 模組中), 259

leaveok() (*curses.window* 的方法), 940
left (*filecmp.dircmp* 的屬性), 482
left() (於 *turtle* 模組中), 1636
left_list (*filecmp.dircmp* 的屬性), 482
left_only (*filecmp.dircmp* 的屬性), 482
LEFTSHIFT (於 *token* 模組中), 2137
LEFTSHIFTEQUAL (於 *token* 模組中), 2138
LEGACY_TRANSACTION_CONTROL (於 *sqlite3* 模組中), 536
len
    built-in function (🄵建函式), 46, 91
len()
    built-in function, 20
length (*xml.dom.NamedNodeMap* 的屬性), 1391
length (*xml.dom.NodeList* 的屬性), 1388
length_hint() (於 *operator* 模組中), 438
--lenient
    mimetypes 命令列選項, 1351
LESS (於 *token* 模組中), 2136
LESSEQUAL (於 *token* 模組中), 2137
level (*logging.Logger* 的屬性), 761
lexical analyzer (詞法分析器), **2236**
LexicalHandler (*xml.sax.handler* 中的類🄵), 1402
lexists() (於 *os.path* 模組中), 470
LF (於 *curses.ascii* 模組中), 955
lgamma() (於 *math* 模組中), 348
libc_ver() (於 *platform* 模組中), 807
LIBRARIES_ASSEMBLY_NAME_PREFIX (於 *msvcrt* 模組中), 2179
library (*ssl.SSLError* 的屬性), 1205
library (於 *dbm.ndbm* 模組中), 530
LibraryLoader (*ctypes* 中的類🄵), 837
license (🄵建變數), 36
LifoQueue (*asyncio* 中的類🄵), 1117
LifoQueue (*queue* 中的類🄵), 1064
light-weight processes (輕量級行程), 1071
limit_denominator() (*fractions.Fraction* 的方法), 383
LimitOverrunError, 1118
line (*bdb.Breakpoint* 的屬性), 1860
LINE (*monitoring event*), 1948
line (*traceback.FrameSummary* 的屬性), 2006
line_buffering (*io.TextIOWrapper* 的屬性), 745
line_num (*csv.csvreader* 的屬性), 624
linear_regression() (於 *statistics* 模組中), 403
line-buffered I/O (行緩衝 I/O), 24
linecache
    module, 492
lineno (*ast.AST* 的屬性), 2096
lineno (*doctest.DocTest* 的屬性), 1739
lineno (*doctest.Example* 的屬性), 1739
lineno (*inspect.FrameInfo* 的屬性), 2029
lineno (*inspect.Traceback* 的屬性), 2029
lineno (*json.JSONDecodeError* 的屬性), 1326
lineno (*netrc.NetrcParseError* 的屬性), 646
lineno (*pyclbr.Class* 的屬性), 2145
lineno (*pyclbr.Function* 的屬性), 2144
lineno (*re.PatternError* 的屬性), 151
lineno (*shlex.shlex* 的屬性), 2194
lineno (*SyntaxError* 的屬性), 114
lineno (*tomllib.TOMLDecodeError* 的屬性), 645
lineno (*traceback.FrameSummary* 的屬性), 2006
lineno (*traceback.TracebackException* 的屬性), 2004
lineno (*tracemalloc.Filter* 的屬性), 1899
lineno (*tracemalloc.Frame* 的屬性), 1900
lineno (*xml.parsers.expat.ExpatError* 的屬性), 1417
lineno() (於 *fileinput* 模組中), 927
LINES, 930, 935
--lines
    calendar 命令列選項, 263
lines (*os.terminal_size* 的屬性), 689
LINES (於 *curses* 模組中), 942
linesep (*email.policy.Policy* 的屬性), 1279
linesep (於 *os* 模組中), 732
lineterminator (*csv.Dialect* 的屬性), 624
LineTooLong, 1468
link() (於 *os* 模組中), 694
linked_to_musl() (於 *test.support* 模組中), 1843
LinkFallbackError, 605
linkname (*tarfile.TarInfo* 的屬性), 610
LinkOutsideDestinationError, 605
--list
    tarfile 命令列選項, 616
    zipfile 命令列選項, 601
List (*ast* 中的類🄵), 2100
list (*pdb command*), 1874
List (*typing* 中的類🄵), 1714
list (🄵建類🄵), 49
list comprehension (串列綜合運算), **2236**
list() (*imaplib.IMAP4* 的方法), 1487
list() (*multiprocessing.managers.SyncManager* 的方法), 1006
list() (*poplib.POP3* 的方法), 1481
list() (*tarfile.TarFile* 的方法), 607
list_all() (於 *concurrent.interpreters* 模組中), 1042
LIST_APPEND (*opcode*), 2160
list_dialects() (於 *csv* 模組中), 620
LIST_EXTEND (*opcode*), 2164
list_folders() (*mailbox.Maildir* 的方法), 1333
list_folders() (*mailbox.MH* 的方法), 1336
ListComp (*ast* 中的類🄵), 2105
listdir() (於 *os* 模組中), 694
listdrives() (於 *os* 模組中), 695
listen() (*socket.socket* 的方法), 1193
listen() (於 *logging.config* 模組中), 780
listen() (於 *turtle* 模組中), 1654
listener (*logging.handlers.QueueHandler* 的屬性), 803
Listener (*multiprocessing.connection* 中的類🄵), 1013
--listfuncs
    trace 命令列選項, 1891
listMethods() (*xmlrpc.client.ServerProxy.system* 的方法), 1532
listmounts() (於 *os* 模組中), 695
listvolumes() (於 *os* 模組中), 695
listxattr() (於 *os* 模組中), 715

list（串列）, **2236**
    object（物件）, 48, 49
    type（型🄵）, operations on（操作於）, 48
Literal (於 *typing* 模組中), 1683
literal_eval() (於 *ast* 模組中), 2125
LiteralString (於 *typing* 模組中), 1679
literals（字面值）
    binary（二進位）, 38
    complex number（🄵數）, 38
    floating-point（浮點數）, 38
    hexadecimal（十六進位）, 38
    integer（整數）, 38
    numeric（數值）, 38
    octal（八進位）, 38
LittleEndianStructure (*ctypes* 中的類🄵), 849
LittleEndianUnion (*ctypes* 中的類🄵), 849
ljust() (*bytearray* 的方法), 73
ljust() (*bytes* 的方法), 73
ljust() (*str* 的方法), 58
LK_LOCK (於 *msvcrt* 模組中), 2177
LK_NBLCK (於 *msvcrt* 模組中), 2177
LK_NBRLCK (於 *msvcrt* 模組中), 2177
LK_RLCK (於 *msvcrt* 模組中), 2177
LK_UNLCK (於 *msvcrt* 模組中), 2177
ll (*pdb command*), 1874
LMTP (*smtplib* 中的類🄵), 1491
ln() (*decimal.Context* 的方法), 369
ln() (*decimal.Decimal* 的方法), 361
LNKTYPE (於 *tarfile* 模組中), 605
Load (*ast* 中的類🄵), 2101
load() (*http.cookiejar.FileCookieJar* 的方法), 1525
load() (*http.cookies.BaseCookie* 的方法), 1519
load() (*pickle.Unpickler* 的方法), 509
load() (*tracemalloc.Snapshot* 的類🄵方法), 1900
load() (於 *json* 模組中), 1323
load() (於 *marshal* 模組中), 525
load() (於 *pickle* 模組中), 507
load() (於 *plistlib* 模組中), 647
load() (於 *tomllib* 模組中), 644
LOAD_ATTR (*opcode*), 2165
LOAD_BUILD_CLASS (*opcode*), 2161
load_cert_chain() (*ssl.SSLContext* 的方法), 1219
LOAD_CLOSURE (*opcode*), 2173
LOAD_COMMON_CONSTANT (*opcode*), 2161
LOAD_CONST (*opcode*), 2163
LOAD_CONST_IMMORTAL (*opcode*), 2172
load_default_certs() (*ssl.SSLContext* 的方法), 1220
LOAD_DEREF (*opcode*), 2167
load_dh_params() (*ssl.SSLContext* 的方法), 1222
load_extension() (*sqlite3.Connection* 的方法), 543
LOAD_FAST (*opcode*), 2166
LOAD_FAST_AND_CLEAR (*opcode*), 2167
LOAD_FAST_BORROW (*opcode*), 2167
LOAD_FAST_BORROW_LOAD_FAST_BORROW (*opcode*), 2167
LOAD_FAST_CHECK (*opcode*), 2167
LOAD_FAST_LOAD_FAST (*opcode*), 2167
LOAD_FROM_DICT_OR_DEREF (*opcode*), 2167
LOAD_FROM_DICT_OR_GLOBALS (*opcode*), 2163
LOAD_GLOBAL (*opcode*), 2166
LOAD_LOCALS (*opcode*), 2163
load_module() (*importlib.abc.FileLoader* 的方法), 2065
load_module() (*importlib.abc.InspectLoader* 的方法), 2064
load_module() (*importlib.abc.Loader* 的方法), 2063
load_module() (*importlib.abc.SourceLoader* 的方法), 2066
load_module() (*importlib.machinery.SourceFileLoader* 的方法), 2070
load_module() (*importlib.machinery.SourcelessFileLoader* 的方法), 2071
load_module() (*zipimport.zipimporter* 的方法), 2052
LOAD_NAME (*opcode*), 2163
load_package_tests() (於 *test.support* 模組中), 1844
LOAD_SMALL_INT (*opcode*), 2163
LOAD_SPECIAL (*opcode*), 2172
LOAD_SUPER_ATTR (*opcode*), 2165
load_verify_locations() (*ssl.SSLContext* 的方法), 1220
Loader (*importlib.abc* 中的類🄵), 2062
loader (*importlib.machinery.ModuleSpec* 的屬性), 2072
loader_state (*importlib.machinery.ModuleSpec* 的屬性), 2072
LoadError, 1522
loader（載入器）, **2236**
LoadFileDialog (*tkinter.filedialog* 中的類🄵), 1595
LoadKey() (於 *winreg* 模組中), 2182
LoadLibrary() (*ctypes.LibraryLoader* 的方法), 838
loads() (於 *json* 模組中), 1324
loads() (於 *marshal* 模組中), 525
loads() (於 *pickle* 模組中), 508
loads() (於 *plistlib* 模組中), 648
loads() (於 *tomllib* 模組中), 645
loads() (於 *xmlrpc.client* 模組中), 1536
loadTestsFromModule() (*unittest.TestLoader* 的方法), 1768
loadTestsFromName() (*unittest.TestLoader* 的方法), 1768
loadTestsFromNames() (*unittest.TestLoader* 的方法), 1768
loadTestsFromTestCase() (*unittest.TestLoader* 的方法), 1768
local (*threading* 中的類🄵), 971
LOCAL_CREDS (於 *socket* 模組中), 1183
LOCAL_CREDS_PERSISTENT (於 *socket* 模組中), 1183
localcontext() (於 *decimal* 模組中), 365
locale
    module, 1569
--locale
    calendar 命令列選項, 262

LOCALE (於 *re* 模組中), 147
locale encoding（區域編碼）, **2236**
localeconv() (於 *locale* 模組中), 1569
LocaleHTMLCalendar (*calendar* 中的類🄵), 259
LocaleTextCalendar (*calendar* 中的類🄵), 259
localize() (於 *locale* 模組中), 1575
localName (*xml.dom.Attr* 的屬性), 1391
localName (*xml.dom.Node* 的屬性), 1387
--locals
    unittest 命令列選項, 1749
locals()
    built-in function, 20
localtime() (於 *email.utils* 模組中), 1316
localtime() (於 *time* 模組中), 750
Locator (*xml.sax.xmlreader* 中的類🄵), 1408
Lock (*asyncio* 中的類🄵), 1106
Lock (*multiprocessing* 中的類🄵), 1000
lock (*sys.thread_info* 的屬性), 1944
Lock (*threading* 中的類🄵), 974
lock() (*mailbox.Babyl* 的方法), 1338
lock() (*mailbox.Mailbox* 的方法), 1332
lock() (*mailbox.Maildir* 的方法), 1335
lock() (*mailbox.mbox* 的方法), 1335
lock() (*mailbox.MH* 的方法), 1337
lock() (*mailbox.MMDF* 的方法), 1338
Lock() (*multiprocessing.managers.SyncManager* 的方法), 1006
LOCK_EX (於 *fcntl* 模組中), 2205
LOCK_NB (於 *fcntl* 模組中), 2205
LOCK_SH (於 *fcntl* 模組中), 2205
LOCK_UN (於 *fcntl* 模組中), 2205
locked() (*_thread.lock* 的方法), 1073
locked() (*asyncio.Condition* 的方法), 1108
locked() (*asyncio.Lock* 的方法), 1106
locked() (*asyncio.Semaphore* 的方法), 1109
locked() (*multiprocessing.Lock* 的方法), 1000
locked() (*multiprocessing.RLock* 的方法), 1001
locked() (*threading.Condition* 的方法), 977
locked() (*threading.Lock* 的方法), 974
locked() (*threading.RLock* 的方法), 976
lockf() (於 *fcntl* 模組中), 2205
lockf() (於 *os* 模組中), 680
locking() (於 *msvcrt* 模組中), 2177
LockType (於 *_thread* 模組中), 1072
log() (*logging.Logger* 的方法), 764
log() (於 *cmath* 模組中), 351
log() (於 *logging* 模組中), 774
log() (於 *math* 模組中), 345
log1p() (於 *math* 模組中), 345
log2() (於 *math* 模組中), 345
log10() (*decimal.Context* 的方法), 369
log10() (*decimal.Decimal* 的方法), 361
log10() (於 *cmath* 模組中), 351
log10() (於 *math* 模組中), 345
LOG_ALERT (於 *syslog* 模組中), 2211
LOG_AUTH (於 *syslog* 模組中), 2211
LOG_AUTHPRIV (於 *syslog* 模組中), 2211
LOG_CONS (於 *syslog* 模組中), 2212
LOG_CRIT (於 *syslog* 模組中), 2211
LOG_CRON (於 *syslog* 模組中), 2211
LOG_DAEMON (於 *syslog* 模組中), 2211
log_date_time_string() (*http.server.BaseHTTPRequestHandler* 的方法), 1515
LOG_DEBUG (於 *syslog* 模組中), 2211
LOG_EMERG (於 *syslog* 模組中), 2211
LOG_ERR (於 *syslog* 模組中), 2211
log_error() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
log_exception() (*wsgiref.handlers.BaseHandler* 的方法), 1432
LOG_FTP (於 *syslog* 模組中), 2211
LOG_INFO (於 *syslog* 模組中), 2211
LOG_INSTALL (於 *syslog* 模組中), 2211
LOG_KERN (於 *syslog* 模組中), 2211
LOG_LAUNCHD (於 *syslog* 模組中), 2211
LOG_LOCAL0 (於 *syslog* 模組中), 2211
LOG_LOCAL1 (於 *syslog* 模組中), 2211
LOG_LOCAL2 (於 *syslog* 模組中), 2211
LOG_LOCAL3 (於 *syslog* 模組中), 2211
LOG_LOCAL4 (於 *syslog* 模組中), 2211
LOG_LOCAL5 (於 *syslog* 模組中), 2211
LOG_LOCAL6 (於 *syslog* 模組中), 2211
LOG_LOCAL7 (於 *syslog* 模組中), 2211
LOG_LPR (於 *syslog* 模組中), 2211
LOG_MAIL (於 *syslog* 模組中), 2211
log_message() (*http.server.BaseHTTPRequestHandler* 的方法), 1515
LOG_NDELAY (於 *syslog* 模組中), 2212
LOG_NETINFO (於 *syslog* 模組中), 2211
LOG_NEWS (於 *syslog* 模組中), 2211
LOG_NOTICE (於 *syslog* 模組中), 2211
LOG_NOWAIT (於 *syslog* 模組中), 2212
LOG_ODELAY (於 *syslog* 模組中), 2212
LOG_PERROR (於 *syslog* 模組中), 2212
LOG_PID (於 *syslog* 模組中), 2212
LOG_RAS (於 *syslog* 模組中), 2211
LOG_REMOTEAUTH (於 *syslog* 模組中), 2211
log_request() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
LOG_SYSLOG (於 *syslog* 模組中), 2211
log_to_stderr() (於 *multiprocessing* 模組中), 1016
LOG_USER (於 *syslog* 模組中), 2211
LOG_UUCP (於 *syslog* 模組中), 2211
LOG_WARNING (於 *syslog* 模組中), 2211
logb() (*decimal.Context* 的方法), 369
logb() (*decimal.Decimal* 的方法), 361
Logger (*logging* 中的類🄵), 761
LoggerAdapter (*logging* 中的類🄵), 773
logging
    module, 759
logging.config
    module, 778
logging.handlers
    module, 790
logging（日🄵）

    Errors（錯誤）, 759
logical_and() (*decimal.Context* 的方法), 369
logical_and() (*decimal.Decimal* 的方法), 361
logical_invert() (*decimal.Context* 的方法), 369
logical_invert() (*decimal.Decimal* 的方法), 361
logical_or() (*decimal.Context* 的方法), 369
logical_or() (*decimal.Decimal* 的方法), 361
logical_xor() (*decimal.Context* 的方法), 369
logical_xor() (*decimal.Decimal* 的方法), 361
login() (*ftplib.FTP* 的方法), 1475
login() (*imaplib.IMAP4* 的方法), 1487
login() (*smtplib.SMTP* 的方法), 1493
login_cram_md5() (*imaplib.IMAP4* 的方法), 1487
login_tty() (於 *os* 模組中), 680
LOGNAME, 672, 926
lognormvariate() (於 *random* 模組中), 388
logout() (*imaplib.IMAP4* 的方法), 1487
LogRecord (*logging* 中的類🄵), 770
LONG_TIMEOUT (於 *test.support* 模組中), 1837
longMessage (*unittest.TestCase* 的屬性), 1763
longname() (於 *curses* 模組中), 932
lookup() (*symtable.SymbolTable* 的方法), 2131
lookup() (*tkinter.ttk.Style* 的方法), 1613
lookup() (於 *codecs* 模組中), 194
lookup() (於 *unicodedata* 模組中), 176
lookup_error() (於 *codecs* 模組中), 198
LookupError, 111
loop_factory (*unittest.IsolatedAsyncioTestCase* 的屬性), 1765
LOOPBACK_TIMEOUT (於 *test.support* 模組中), 1837
loop（🄵圈）
    over mutable sequence（於可變序列）, 46
lower() (*bytearray* 的方法), 78
lower() (*bytes* 的方法), 78
lower() (*str* 的方法), 58
LPAR (於 *token* 模組中), 2136
lpAttributeList (*subprocess.STARTUPINFO* 的屬性), 1055
lru_cache() (於 *functools* 模組中), 426
lseek() (於 *os* 模組中), 680
LShift (*ast* 中的類🄵), 2102
lshift() (於 *operator* 模組中), 437
LSQB (於 *token* 模組中), 2136
lstat() (*pathlib.Path* 的方法), 457
lstat() (於 *os* 模組中), 695
lstrip() (*bytearray* 的方法), 73
lstrip() (*bytes* 的方法), 73
lstrip() (*str* 的方法), 58
lsub() (*imaplib.IMAP4* 的方法), 1487
Lt (*ast* 中的類🄵), 2103
lt() (於 *operator* 模組中), 435
lt() (於 *turtle* 模組中), 1636
LtE (*ast* 中的類🄵), 2103
LWPCookieJar (*http.cookiejar* 中的類🄵), 1525
lzma
    module, 585
LZMACompressor (*lzma* 中的類🄵), 587
LZMADecompressor (*lzma* 中的類🄵), 588
LZMAError, 585
LZMAFile (*lzma* 中的類🄵), 586

M

-m
    ast 命令列選項, 2129
    calendar 命令列選項, 263
    cProfile 命令列選項, 1879
    pdb 命令列選項, 1869
    pickletools 命令列選項, 2174
    trace 命令列選項, 1891
    zipapp 命令列選項, 1914
M (於 *re* 模組中), 147
mac_ver() (於 *platform* 模組中), 806
machine() (於 *platform* 模組中), 804
macros (*netrc.netrc* 的屬性), 647
MADV_AUTOSYNC (於 *mmap* 模組中), 1260
MADV_CORE (於 *mmap* 模組中), 1260
MADV_DODUMP (於 *mmap* 模組中), 1260
MADV_DOFORK (於 *mmap* 模組中), 1260
MADV_DONTDUMP (於 *mmap* 模組中), 1260
MADV_DONTFORK (於 *mmap* 模組中), 1260
MADV_DONTNEED (於 *mmap* 模組中), 1260
MADV_FREE (於 *mmap* 模組中), 1260
MADV_FREE_REUSABLE (於 *mmap* 模組中), 1260
MADV_FREE_REUSE (於 *mmap* 模組中), 1260
MADV_HUGEPAGE (於 *mmap* 模組中), 1260
MADV_HWPOISON (於 *mmap* 模組中), 1260
MADV_MERGEABLE (於 *mmap* 模組中), 1260
MADV_NOCORE (於 *mmap* 模組中), 1260
MADV_NOHUGEPAGE (於 *mmap* 模組中), 1260
MADV_NORMAL (於 *mmap* 模組中), 1260
MADV_NOSYNC (於 *mmap* 模組中), 1260
MADV_PROTECT (於 *mmap* 模組中), 1260
MADV_RANDOM (於 *mmap* 模組中), 1260
MADV_REMOVE (於 *mmap* 模組中), 1260
MADV_SEQUENTIAL (於 *mmap* 模組中), 1260
MADV_SOFT_OFFLINE (於 *mmap* 模組中), 1260
MADV_UNMERGEABLE (於 *mmap* 模組中), 1260
MADV_WILLNEED (於 *mmap* 模組中), 1260
madvise() (*mmap.mmap* 的方法), 1258
magic
    method（方法）, 2237
magic method（魔術方法）, **2237**
MAGIC_NUMBER (於 *importlib.util* 模組中), 2073
MagicMock (*unittest.mock* 中的類🄵), 1805
mailbox
    module, 1329
Mailbox (*mailbox* 中的類🄵), 1329
mailcap
    module, 2221
Maildir (*mailbox* 中的類🄵), 1332
MaildirMessage (*mailbox* 中的類🄵), 1339
--main
    zipapp 命令列選項, 1914
main() (於 *site* 模組中), 2045
main() (於 *unittest* 模組中), 1773
main_thread() (於 *threading* 模組中), 968

mainloop() (於 *turtle* 模組中), 1656
maintype (*email.headerregistry.ContentTypeHeader* 的屬性), 1289
major (*email.headerregistry.MIMEVersionHeader* 的屬性), 1289
major() (於 *os* 模組中), 697
make_alternative() (*email.message.EmailMessage* 的方法), 1270
make_archive() (於 *shutil* 模組中), 499
make_bad_fd() (於 *test.support.os_helper* 模組中), 1850
MAKE_CELL (*opcode*), 2167
make_cookies() (*http.cookiejar.CookieJar* 的方法), 1524
make_dataclass() (於 *dataclasses* 模組中), 1974
make_file() (*difflib.HtmlDiff* 的方法), 162
MAKE_FUNCTION (*opcode*), 2169
make_header() (於 *email.header* 模組中), 1313
make_legacy_pyc() (於 *test.support.import_helper* 模組中), 1851
make_mixed() (*email.message.EmailMessage* 的方法), 1271
make_msgid() (於 *email.utils* 模組中), 1316
make_parser() (於 *xml.sax* 模組中), 1400
make_pkg() (於 *test.support.script_helper* 模組中), 1847
make_related() (*email.message.EmailMessage* 的方法), 1270
make_script() (於 *test.support.script_helper* 模組中), 1847
make_server() (於 *wsgiref.simple_server* 模組中), 1428
make_table() (*difflib.HtmlDiff* 的方法), 162
make_zip_pkg() (於 *test.support.script_helper* 模組中), 1847
make_zip_script() (於 *test.support.script_helper* 模組中), 1847
makedev() (於 *os* 模組中), 697
makedirs() (於 *os* 模組中), 696
makeelement() (*xml.etree.ElementTree.Element* 的方法), 1378
makefile() (*socket.socket* 的方法), 1193
makeLogRecord() (於 *logging* 模組中), 775
makePickle() (*logging.handlers.SocketHandler* 的方法), 796
makeRecord() (*logging.Logger* 的方法), 765
makeSocket() (*logging.handlers.DatagramHandler* 的方法), 797
makeSocket() (*logging.handlers.SocketHandler* 的方法), 796
maketrans() (*bytearray* 的🄵態方法), 72
maketrans() (*bytes* 的🄵態方法), 72
maketrans() (*str* 的🄵態方法), 59
MalformedHeaderDefect, 1286
manager (*logging.LoggerAdapter* 的屬性), 773
mangle_from_ (*email.policy.Compat32* 的屬性), 1284
mangle_from_ (*email.policy.Policy* 的屬性), 1280
MANPAGER, 1720
mant_dig (*sys.float_info* 的屬性), 1928
map()
    built-in function, 20
map() (*concurrent.futures.Executor* 的方法), 1032
map() (*multiprocessing.pool.Pool* 的方法), 1011
map() (*tkinter.ttk.Style* 的方法), 1613
MAP_32BIT (於 *mmap* 模組中), 1260
MAP_ADD (*opcode*), 2160
MAP_ALIGNED_SUPER (於 *mmap* 模組中), 1260
MAP_ANON (於 *mmap* 模組中), 1260
MAP_ANONYMOUS (於 *mmap* 模組中), 1260
map_async() (*multiprocessing.pool.Pool* 的方法), 1011
MAP_CONCEAL (於 *mmap* 模組中), 1260
MAP_DENYWRITE (於 *mmap* 模組中), 1260
MAP_EXECUTABLE (於 *mmap* 模組中), 1260
MAP_HASSEMAPHORE (於 *mmap* 模組中), 1260
MAP_JIT (於 *mmap* 模組中), 1260
MAP_NOCACHE (於 *mmap* 模組中), 1260
MAP_NOEXTEND (於 *mmap* 模組中), 1260
MAP_NORESERVE (於 *mmap* 模組中), 1260
MAP_POPULATE (於 *mmap* 模組中), 1260
MAP_PRIVATE (於 *mmap* 模組中), 1260
MAP_RESILIENT_CODESIGN (於 *mmap* 模組中), 1260
MAP_RESILIENT_MEDIA (於 *mmap* 模組中), 1260
MAP_SHARED (於 *mmap* 模組中), 1260
MAP_STACK (於 *mmap* 模組中), 1260
map_table_b2() (於 *stringprep* 模組中), 179
map_table_b3() (於 *stringprep* 模組中), 179
map_to_type() (*email.headerregistry.HeaderRegistry* 的方法), 1290
MAP_TPRO (於 *mmap* 模組中), 1260
MAP_TRANSLATED_ALLOW_EXECUTE (於 *mmap* 模組中), 1260
MAP_UNIX03 (於 *mmap* 模組中), 1260
mapLogRecord() (*logging.handlers.HTTPHandler* 的方法), 801
Mapping (*collections.abc* 中的類🄵), 284
Mapping (*typing* 中的類🄵), 1717
MappingProxyType (*types* 中的類🄵), 307
MappingView (*collections.abc* 中的類🄵), 284
MappingView (*typing* 中的類🄵), 1717
mapping（對映）, **2237**
    object（物件）, 91
    type（型🄵）, operations on（操作於）, 91
mapPriority() (*logging.handlers.SysLogHandler* 的方法), 799
maps (*collections.ChainMap* 的屬性), 264
MARCH (於 *calendar* 模組中), 261
markcoroutinefunction() (於 *inspect* 模組中), 2019
marshal
    module, 524
marshalling
    objects（物件）, 505
masking（遮罩）
    operations（操作）, 40
master (*tkinter.Tk* 的屬性), 1581

Match (*ast* 中的類F), 2114
Match (*re* 中的類F), 153
Match (*typing* 中的類F), 1716
match() (*pathlib.PurePath* 的方法), 451
match() (*re.Pattern* 的方法), 152
match() (於 *re* 模組中), 148
match_case (*ast* 中的類F), 2114
MATCH_CLASS (*opcode*), 2170
MATCH_KEYS (*opcode*), 2162
MATCH_MAPPING (*opcode*), 2162
MATCH_SEQUENCE (*opcode*), 2162
match_value() (*test.support.Matcher* 的方法), 1845
MatchAs (*ast* 中的類F), 2118
MatchClass (*ast* 中的類F), 2117
Matcher (*test.support* 中的類F), 1845
matches() (*test.support.Matcher* 的方法), 1845
MatchMapping (*ast* 中的類F), 2116
MatchOr (*ast* 中的類F), 2118
MatchSequence (*ast* 中的類F), 2115
MatchSingleton (*ast* 中的類F), 2115
MatchStar (*ast* 中的類F), 2116
MatchValue (*ast* 中的類F), 2114
math
    module, 340
    模組, 39
math（數學）
    module（模組）, 353
matmul() (於 *operator* 模組中), 437
MatMult (*ast* 中的類F), 2102
max
    built-in function（F建函式）, 46
max (*datetime.date* 的屬性), 221
max (*datetime.datetime* 的屬性), 228
max (*datetime.time* 的屬性), 236
max (*datetime.timedelta* 的屬性), 217
max (*sys.float_info* 的屬性), 1928
MAX (於 *uuid* 模組中), 1500
max()
    built-in function, 21
max() (*decimal.Context* 的方法), 369
max() (*decimal.Decimal* 的方法), 361
max_10_exp (*sys.float_info* 的屬性), 1928
max_children (*socketserver.ThreadingMixIn* 的屬性), 1504
max_count (*email.headerregistry.BaseHeader* 的屬性), 1287
MAX_EMAX (於 *decimal* 模組中), 371
max_exp (*sys.float_info* 的屬性), 1928
MAX_INTERPOLATION_DEPTH (於 *configparser* 模組中), 642
max_line_length (*email.policy.Policy* 的屬性), 1279
max_lines (*textwrap.TextWrapper* 的屬性), 176
max_mag() (*decimal.Context* 的方法), 369
max_mag() (*decimal.Decimal* 的方法), 361
max_memuse (於 *test.support* 模組中), 1838
MAX_PREC (於 *decimal* 模組中), 371
max_prefixlen (*ipaddress.IPv4Address* 的屬性), 1544
max_prefixlen (*ipaddress.IPv4Network* 的屬性), 1550
max_prefixlen (*ipaddress.IPv6Address* 的屬性), 1547
max_prefixlen (*ipaddress.IPv6Network* 的屬性), 1552
MAX_Py_ssize_t (於 *test.support* 模組中), 1838
maxarray (*reprlib.Repr* 的屬性), 317
maxdeque (*reprlib.Repr* 的屬性), 317
maxdict (*reprlib.Repr* 的屬性), 317
maxDiff (*unittest.TestCase* 的屬性), 1764
maxfrozenset (*reprlib.Repr* 的屬性), 317
MAXIMUM_SUPPORTED (*ssl.TLSVersion* 的屬性), 1214
maximum_version (*ssl.SSLContext* 的屬性), 1225
maxlen (*collections.deque* 的屬性), 270
maxlevel (*reprlib.Repr* 的屬性), 317
maxlist (*reprlib.Repr* 的屬性), 317
maxlong (*reprlib.Repr* 的屬性), 317
maxother (*reprlib.Repr* 的屬性), 317
maxset (*reprlib.Repr* 的屬性), 317
maxsize (*asyncio.Queue* 的屬性), 1115
maxsize (於 *sys* 模組中), 1936
maxstring (*reprlib.Repr* 的屬性), 317
maxtuple (*reprlib.Repr* 的屬性), 317
maxunicode (於 *sys* 模組中), 1936
MAXYEAR (於 *datetime* 模組中), 214
MAY (於 *calendar* 模組中), 261
MB_ICONASTERISK (於 *winsound* 模組中), 2190
MB_ICONERROR (於 *winsound* 模組中), 2190
MB_ICONEXCLAMATION (於 *winsound* 模組中), 2190
MB_ICONHAND (於 *winsound* 模組中), 2190
MB_ICONINFORMATION (於 *winsound* 模組中), 2190
MB_ICONQUESTION (於 *winsound* 模組中), 2190
MB_ICONSTOP (於 *winsound* 模組中), 2190
MB_ICONWARNING (於 *winsound* 模組中), 2190
MB_OK (於 *winsound* 模組中), 2190
mbox (*mailbox* 中的類F), 1335
mboxMessage (*mailbox* 中的類F), 1341
md5() (於 *hashlib* 模組中), 652
mean (*statistics.NormalDist* 的屬性), 404
mean() (於 *statistics* 模組中), 395
measure() (*tkinter.font.Font* 的方法), 1592
median (*statistics.NormalDist* 的屬性), 404
median() (於 *statistics* 模組中), 397
median_grouped() (於 *statistics* 模組中), 398
median_high() (於 *statistics* 模組中), 398
median_low() (於 *statistics* 模組中), 398
member() (於 *enum* 模組中), 333
member_names (*enum.EnumDict* 的屬性), 331
MemberDescriptorType (於 *types* 模組中), 307
memfd_create() (於 *os* 模組中), 710
memmove() (於 *ctypes* 模組中), 844
--memo
    pickletools 命令列選項, 2174
MemoryBIO (*ssl* 中的類F), 1234
MemoryError, 112
MemoryHandler (*logging.handlers* 中的類F), 801
memoryview (F建類F), 82

memoryview_at() (於 *ctypes* 模組中), 845
memoryview（記憶體視圖）
    object（物件）, 66
memset() (於 *ctypes* 模組中), 844
merge() (於 *heapq* 模組中), 287
message (*BaseExceptionGroup* 的屬性), 118
Message (*email.message* 中的類🄵), 1301
Message (*mailbox* 中的類🄵), 1339
Message (*tkinter.messagebox* 中的類🄵), 1595
message digest（訊息摘要）、MD5, 651
message_factory (*email.policy.Policy* 的屬性), 1280
message_from_binary_file() (於 *email* 模組中), 1274
message_from_bytes() (於 *email* 模組中), 1274
message_from_file() (於 *email* 模組中), 1274
message_from_string() (於 *email* 模組中), 1274
MessageBeep() (於 *winsound* 模組中), 2188
MessageClass (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
MessageDefect, 1285
MessageError, 1285
MessageParseError, 1285
messages (於 *xml.parsers.expat.errors* 模組中), 1418
meta path finder（元路徑尋檢器）, **2237**
meta() (於 *curses* 模組中), 932
meta_path (於 *sys* 模組中), 1936
metaclass（元類🄵）, **2237**
metadata() (於 *importlib.metadata* 模組中), 2086
--metadata-encoding
    zipfile 命令列選項, 601
MetaPathFinder (*importlib.abc* 中的類🄵), 2062
metavar (*optparse.Option* 的屬性), 912
MetavarTypeHelpFormatter (*argparse* 中的類🄵), 859
method (*urllib.request.Request* 的屬性), 1441
method resolution order(方法解析順序), **2237**
method_calls (*unittest.mock.Mock* 的屬性), 1785
methodcaller() (於 *operator* 模組中), 439
MethodDescriptorType (於 *types* 模組中), 306
methodHelp() (*xmlrpc.client.ServerProxy.system* 的方法), 1532
methods (*pyclbr.Class* 的屬性), 2145
methodSignature() (*xmlrpc.client.ServerProxy.system* 的方法), 1532
methods（方法）
    bytearray（位元組陣列）, 69
    bytes（位元組）, 69
    string（字串）, 54
MethodType (於 *types* 模組中), 306
MethodWrapperType (於 *types* 模組中), 306
method（方法）, **2237**
    magic, 2237
    object（物件）, 103
    special, 2241
metrics() (*tkinter.font.Font* 的方法), 1592
MFD_ALLOW_SEALING (於 *os* 模組中), 710
MFD_CLOEXEC (於 *os* 模組中), 710
MFD_HUGE_1GB (於 *os* 模組中), 710
MFD_HUGE_1MB (於 *os* 模組中), 710
MFD_HUGE_2GB (於 *os* 模組中), 710
MFD_HUGE_2MB (於 *os* 模組中), 710
MFD_HUGE_8MB (於 *os* 模組中), 710
MFD_HUGE_16GB (於 *os* 模組中), 710
MFD_HUGE_16MB (於 *os* 模組中), 710
MFD_HUGE_32MB (於 *os* 模組中), 710
MFD_HUGE_64KB (於 *os* 模組中), 710
MFD_HUGE_256MB (於 *os* 模組中), 710
MFD_HUGE_512KB (於 *os* 模組中), 710
MFD_HUGE_512MB (於 *os* 模組中), 710
MFD_HUGE_MASK (於 *os* 模組中), 710
MFD_HUGE_SHIFT (於 *os* 模組中), 710
MFD_HUGETLB (於 *os* 模組中), 710
MH (*mailbox* 中的類🄵), 1336
MHMessage (*mailbox* 中的類🄵), 1343
microsecond (*datetime.datetime* 的屬性), 228
microsecond (*datetime.time* 的屬性), 236
microseconds (*datetime.timedelta* 的屬性), 217
MIME
    base64 encoding（base64 編碼）, 1352
    content type（🄵容類型）, 1348
    headers（標頭）, 1348
    quoted-printable encoding(可列印字元編碼), 1358
MIMEApplication (*email.mime.application* 中的類🄵), 1309
MIMEAudio (*email.mime.audio* 中的類🄵), 1309
MIMEBase (*email.mime.base* 中的類🄵), 1308
MIMEImage (*email.mime.image* 中的類🄵), 1310
MIMEMessage (*email.mime.message* 中的類🄵), 1310
MIMEMultipart (*email.mime.multipart* 中的類🄵), 1309
MIMENonMultipart (*email.mime.nonmultipart* 中的類🄵), 1309
MIMEPart (*email.message* 中的類🄵), 1272
MIMEText (*email.mime.text* 中的類🄵), 1310
mimetypes
    module, 1348
MimeTypes (*mimetypes* 中的類🄵), 1350
mimetypes 命令列選項
    -e, 1351
    --extension, 1351
    -h, 1351
    --help, 1351
    -l, 1351
    --lenient, 1351
MIMEVersionHeader (*email.headerregistry* 中的類🄵), 1288
min
    built-in function（🄵建函式）, 46
min (*datetime.date* 的屬性), 221
min (*datetime.datetime* 的屬性), 228
min (*datetime.time* 的屬性), 236
min (*datetime.timedelta* 的屬性), 217
min (*sys.float_info* 的屬性), 1928
min()

    built-in function, 21
min() (*decimal.Context* 的方法), 369
min() (*decimal.Decimal* 的方法), 362
min_10_exp (*sys.float_info* 的屬性), 1928
MIN_EMIN (於 *decimal* 模組中), 371
MIN_ETINY (於 *decimal* 模組中), 371
min_exp (*sys.float_info* 的屬性), 1928
min_mag() (*decimal.Context* 的方法), 369
min_mag() (*decimal.Decimal* 的方法), 362
min_match (*compression.zstd.CompressionParameter* 的屬性), 570
MINEQUAL (於 *token* 模組中), 2137
MINIMUM_SUPPORTED (*ssl.TLSVersion* 的屬性), 1214
minimum_version (*ssl.SSLContext* 的屬性), 1225
minor (*email.headerregistry.MIMEVersionHeader* 的屬性), 1289
minor() (於 *os* 模組中), 697
MINUS (於 *token* 模組中), 2136
minus() (*decimal.Context* 的方法), 369
minute (*datetime.datetime* 的屬性), 228
minute (*datetime.time* 的屬性), 236
MINYEAR (於 *datetime* 模組中), 214
mirrored() (於 *unicodedata* 模組中), 178
misc_header (*cmd.Cmd* 的屬性), 961
MisplacedEnvelopeHeaderDefect, 1285
--missing
    trace 命令列選項, 1891
MISSING (*contextvars.Token* 的屬性), 1069
MISSING (於 *dataclasses* 模組中), 1975
MISSING (於 *sys.monitoring* 模組中), 1950
MISSING_C_DOCSTRINGS (於 *test.support* 模組中), 1838
missing_compiler_executable() (於 *test.support* 模組中), 1844
MissingHeaderBodySeparatorDefect, 1285
MissingSectionHeaderError, 644
mkd() (*ftplib.FTP* 的方法), 1477
mkdir() (*pathlib.Path* 的方法), 463
mkdir() (於 *os* 模組中), 696
mkdir() (*zipfile.ZipFile* 的方法), 596
mkdtemp() (於 *tempfile* 模組中), 485
mkfifo() (於 *os* 模組中), 696
mknod() (於 *os* 模組中), 697
mkstemp() (於 *tempfile* 模組中), 485
mktemp() (於 *tempfile* 模組中), 487
mktime() (於 *time* 模組中), 750
mktime_tz() (於 *email.utils* 模組中), 1317
mlsd() (*ftplib.FTP* 的方法), 1477
mmap
    module, 1256
mmap (*mmap* 中的類F), 1256
MMDF (*mailbox* 中的類F), 1338
MMDFMessage (*mailbox* 中的類F), 1345
Mock (*unittest.mock* 中的類F), 1779
mock_add_spec() (*unittest.mock.Mock* 的方法), 1782
mock_calls (*unittest.mock.Mock* 的屬性), 1785
mock_open() (於 *unittest.mock* 模組中), 1810
Mod (*ast* 中的類F), 2102
mod() (於 *operator* 模組中), 437
--mode
    ast 命令列選項, 2129
mode (*bz2.BZ2File* 的屬性), 582
mode (*compression.zstd.ZstdFile* 的屬性), 565
mode (*gzip.GzipFile* 的屬性), 579
mode (*io.FileIO* 的屬性), 741
mode (*lzma.LZMAFile* 的屬性), 586
mode (*statistics.NormalDist* 的屬性), 404
mode (*tarfile.TarInfo* 的屬性), 610
mode() (於 *statistics* 模組中), 399
mode() (於 *turtle* 模組中), 1656
modes (模式)
    file (檔案), 22
modf() (於 *math* 模組中), 343
modified() (*urllib.robotparser.RobotFileParser* 的方法), 1462
modify() (*select.devpoll* 的方法), 1239
modify() (*select.epoll* 的方法), 1240
modify() (*selectors.BaseSelector* 的方法), 1245
modify() (*select.poll* 的方法), 1241
module
    __future__, 2010
    __main__, 1958
    _thread, 1071
    _tkinter, 1582
    abc, 1994
    aifc, 2219
    annotationlib, 2035
    argparse, 855
    array, 293
    ast, 2093
    asynchat, 2219
    asyncio, 1075
    asyncore, 2219
    atexit, 1999
    audioop, 2220
    base64, 1352
    bdb, 1859
    binascii, 1356
    bisect, 290
    builtins, 1957
    bz2, 581
    calendar, 255
    cgi, 2220
    cgitb, 2220
    chunk, 2220
    cmath, 349
    cmd, 959
    code, 2047
    codecs, 194
    codeop, 2049
    collections, 263
    collections.abc, 280
    colorsys, 1560
    compileall, 2147
    compression.zstd, 563
    concurrent.futures, 1031

concurrent.interpreters, 1039
configparser, 626
contextlib, 1980
contextvars, 1067
copy, 309
copyreg, 520
cProfile, 1880
crypt, 2220
csv, 619
ctypes, 817
curses, 928
curses.ascii, 954
curses.panel, 958
curses.textpad, 953
dataclasses, 1969
datetime, 213
dbm, 525
dbm.dumb, 530
dbm.gnu, 528
dbm.ndbm, 529
dbm.sqlite3, 527
decimal, 353
difflib, 161
dis, 2151
distutils, 2220
doctest, 1724
email, 1263
email.charset, 1313
email.contentmanager, 1292
email.encoders, 1315
email.errors, 1285
email.generator, 1275
email.header, 1311
email.headerregistry, 1286
email.iterators, 1318
email.message, 1264
email.mime, 1308
email.mime.application, 1309
email.mime.audio, 1309
email.mime.base, 1308
email.mime.image, 1310
email.mime.message, 1310
email.mime.multipart, 1309
email.mime.nonmultipart, 1309
email.mime.text, 1310
email.parser, 1272
email.policy, 1278
email.utils, 1316
encodings, 209
encodings.idna, 210
encodings.mbcs, 211
encodings.utf_8_sig, 211
ensurepip, 1903
enum, 319
errno, 808
faulthandler, 1864
fcntl, 2203
filecmp, 481
fileinput, 926
fnmatch, 490
fractions, 381
ftplib, 1474
functools, 425
gc, 2012
getopt, 2215
getpass, 925
gettext, 1561
glob, 488
graphlib, 334
grp, 2198
gzip, 577
hashlib, 651
heapq, 286
hmac, 662
html, 1359
html.entities, 1364
html.parser, 1360
http, 1463
http.client, 1466
http.cookiejar, 1522
http.cookies, 1518
http.server, 1511
idlelib, 1628
imaplib, 1483
imghdr, 2221
imp, 2221
importlib, 2059
importlib.abc, 2062
importlib.machinery, 2068
importlib.metadata, 2083
importlib.resources, 2079
importlib.resources.abc, 2082
importlib.util, 2073
inspect, 2016
io, 734
ipaddress, 1543
itertools, 409
json, 1319
json.tool, 1328
keyword, 2139
linecache, 492
locale, 1569
logging, 759
logging.config, 778
logging.handlers, 790
lzma, 585
mailbox, 1329
mailcap, 2221
marshal, 524
math, 340
mimetypes, 1348
mmap, 1256
modulefinder, 2055
msilib, 2221
msvcrt, 2177
multiprocessing, 982

multiprocessing.connection, 1013
multiprocessing.dummy, 1017
multiprocessing.managers, 1004
multiprocessing.pool, 1010
multiprocessing.shared_memory, 1026
multiprocessing.sharedctypes, 1002
netrc, 646
nis, 2221
nntplib, 2221
numbers, 337
operator, 435
optparse, 898
os, 667
os.path, 468
ossaudiodev, 2222
pathlib, 443
pathlib.types, 468
pdb, 1868
pickle, 505
pickletools, 2174
pipes, 2222
pkgutil, 2053
platform, 804
plistlib, 647
poplib, 1480
posix, 2196
pprint, 311
profile, 1880
pstats, 1881
pty, 2201
pwd, 2197
py_compile, 2145
pyclbr, 2144
pydoc, 1720
queue, 1064
quopri, 1358
random, 384
re, 139
readline, 180
reprlib, 316
resource, 2206
rlcompleter, 184
runpy, 2057
sched, 1063
secrets, 663
select, 1237
selectors, 1243
shelve, 521
shlex, 2191
shutil, 492
signal, 1247
site, 2043
sitecustomize, 2044
smtpd, 2222
smtplib, 1490
sndhdr, 2222
socket, 1173
socketserver, 1503
spwd, 2222
sqlite3, 531
ssl, 1202
stat, 475
statistics, 393
string, 123
stringprep, 179
string.templatelib, 134
struct, 187
subprocess, 1044
sunau, 2222
symtable, 2130
sys, 1919
sysconfig, 1951
syslog, 2210
sys.monitoring, 1946
tabnanny, 2143
tarfile, 602
telnetlib, 2223
tempfile, 483
termios, 2199
test, 1834
test.regrtest, 1836
test.support, 1836
test.support.bytecode_helper, 1847
test.support.import_helper, 1851
test.support.os_helper, 1849
test.support.script_helper, 1846
test.support.socket_helper, 1846
test.support.threading_helper, 1848
test.support.warnings_helper, 1852
textwrap, 173
threading, 965
time, 748
timeit, 1886
tkinter, 1579
tkinter.colorchooser, 1591
tkinter.commondialog, 1595
tkinter.dnd, 1598
tkinter.filedialog, 1593
tkinter.font, 1591
tkinter.messagebox, 1595
tkinter.scrolledtext, 1597
tkinter.simpledialog, 1592
tkinter.ttk, 1599
token, 2134
tokenize, 2139
tomllib, 644
trace, 1890
traceback, 2000
tracemalloc, 1893
tty, 2200
turtle, 1628
turtledemo, 1663
types, 303
typing, 1667
unicodedata, 176
unittest, 1747

unittest.mock, 1777
urllib, 1435
urllib.error, 1461
urllib.parse, 1452
urllib.request, 1435
urllib.response, 1452
urllib.robotparser, 1462
usercustomize, 2044
uu, 2223
uuid, 1497
venv, 1905
warnings, 1962
wave, 1557
weakref, 296
webbrowser, 1423
winreg, 2180
winsound, 2188
wsgiref, 1426
wsgiref.handlers, 1430
wsgiref.headers, 1428
wsgiref.simple_server, 1428
wsgiref.types, 1433
wsgiref.util, 1426
wsgiref.validate, 1429
xdrlib, 2223
xml, 1364
xml.dom, 1384
xml.dom.minidom, 1394
xml.dom.pulldom, 1398
xml.etree.ElementInclude, 1376
xml.etree.ElementTree, 1365
xml.parsers.expat, 1411
xml.parsers.expat.errors, 1418
xml.parsers.expat.model, 1418
xmlrpc, 1530
xmlrpc.client, 1530
xmlrpc.server, 1537
xml.sax, 1400
xml.sax.handler, 1401
xml.sax.saxutils, 1406
xml.sax.xmlreader, 1407
zipapp, 1914
zipfile, 591
zipimport, 2051
zlib, 574
zoneinfo, 251
Module (*ast* 中的類🄵), 2097
module (*pyclbr.Class* 的屬性), 2145
module (*pyclbr.Function* 的屬性), 2144
MODULE (*symtable.SymbolTableType* 的屬性), 2130
Module browser（模組🄵覽器）, 1617
module spec（模組規格）, **2237**
module_from_spec() (於 *importlib.util* 模組中), 2075
modulefinder
    module, 2055
ModuleFinder (*modulefinder* 中的類🄵), 2056
ModuleInfo (*pkgutil* 中的類🄵), 2053
ModuleNotFoundError, 111
modules (*modulefinder.ModuleFinder* 的屬性), 2056
modules (於 *sys* 模組中), 1937
modules_cleanup() (於 *test.support.import_helper* 模組中), 1851
modules_setup() (於 *test.support.import_helper* 模組中), 1851
ModuleSpec (*importlib.machinery* 中的類🄵), 2072
ModuleType (*types* 中的類🄵), 306
module（模組）, **2237**
    __main__, 2057, 2058
    _locale, 1569
    base64, 1356
    bdb, 1868
    builtins（🄵建）, 32
    cmd, 1868
    copy（🄵🄵）, 520
    dbm.gnu, 522
    dbm.ndbm, 522
    errno, 112
    glob, 490
    math（數學）, 353
    os, 2196
    pickle, 310, 520, 521, 524
    pty, 683
    pwd, 470
    pyexpat, 1411
    re, 490
    search（搜尋）path（路徑）, 492, 1937, 2043
    shelve, 524
    signal（訊號）, 1073
    socket, 1423
    stat, 702
    struct, 1198
    sys, 24
    urllib.request, 1466
modulus (*sys.hash_info* 的屬性), 1932
MON_1 (於 *locale* 模組中), 1571
MON_2 (於 *locale* 模組中), 1571
MON_3 (於 *locale* 模組中), 1571
MON_4 (於 *locale* 模組中), 1571
MON_5 (於 *locale* 模組中), 1571
MON_6 (於 *locale* 模組中), 1571
MON_7 (於 *locale* 模組中), 1571
MON_8 (於 *locale* 模組中), 1571
MON_9 (於 *locale* 模組中), 1571
MON_10 (於 *locale* 模組中), 1571
MON_11 (於 *locale* 模組中), 1571
MON_12 (於 *locale* 模組中), 1571
MONDAY (於 *calendar* 模組中), 260
monotonic() (於 *time* 模組中), 750
monotonic_ns() (於 *time* 模組中), 751
month
    calendar 命令列選項, 263
Month (*calendar* 中的類🄵), 261
month (*calendar.IllegalMonthError* 的屬性), 261
month (*datetime.date* 的屬性), 221
month (*datetime.datetime* 的屬性), 228

month() (於 calendar 模組中), 260
month_abbr (於 calendar 模組中), 261
month_name (於 calendar 模組中), 260
monthcalendar() (於 calendar 模組中), 260
monthdatescalendar() (calendar.Calendar 的方法), 256
monthdays2calendar() (calendar.Calendar 的方法), 257
monthdayscalendar() (calendar.Calendar 的方法), 257
monthrange() (於 calendar 模組中), 260
--months
    calendar 命令列選項, 263
Morsel (http.cookies 中的類🄵), 1520
most_common() (collections.Counter 的方法), 267
mouseinterval() (於 curses 模組中), 932
mousemask() (於 curses 模組中), 932
move() (curses.panel.Panel 的方法), 959
move() (curses.window 的方法), 940
move() (mmap.mmap 的方法), 1259
move() (pathlib.Path 的方法), 464
move() (tkinter.ttk.Treeview 的方法), 1611
move() (於 shutil 模組中), 496
move_into() (pathlib.Path 的方法), 465
move_to_end() (collections.OrderedDict 的方法), 277
MozillaCookieJar (http.cookiejar 中的類🄵), 1525
MRO, **2237**
msg (http.client.HTTPResponse 的屬性), 1472
msg (json.JSONDecodeError 的屬性), 1326
msg (netrc.NetrcParseError 的屬性), 646
msg (re.PatternError 的屬性), 151
msg (tomllib.TOMLDecodeError 的屬性), 645
msg (traceback.TracebackException 的屬性), 2004
msilib
    module, 2221
msvcrt
    module, 2177
mtime (gzip.GzipFile 的屬性), 579
mtime (tarfile.TarInfo 的屬性), 610
mtime() (urllib.robotparser.RobotFileParser 的方法), 1462
mul() (於 operator 模組中), 437
Mult (ast 中的類🄵), 2102
MultiCall (xmlrpc.client 中的類🄵), 1536
MULTILINE (於 re 模組中), 147
MultilineContinuationError, 644
multimode() (於 statistics 模組中), 399
MultipartConversionError, 1285
MultipartInvariantViolationDefect, 1286
multiply() (decimal.Context 的方法), 369
multiprocessing
    module, 982
multiprocessing.connection
    module, 1013
multiprocessing.dummy
    module, 1017
multiprocessing.Manager()
    built-in function, 1004
multiprocessing.managers
    module, 1004
multiprocessing.pool
    module, 1010
multiprocessing.shared_memory
    module, 1026
multiprocessing.sharedctypes
    module, 1002
mutable sequence (可變序列)
    loop over (🄵圈), 46
MutableMapping (collections.abc 中的類🄵), 284
MutableMapping (typing 中的類🄵), 1717
MutableSequence (collections.abc 中的類🄵), 283
MutableSequence (typing 中的類🄵), 1717
MutableSet (collections.abc 中的類🄵), 284
MutableSet (typing 中的類🄵), 1717
mutable (可變物件), **2237**
mutable (可變)
    sequence (序列) type (型🄵), 48
mvderwin() (curses.window 的方法), 940
mvwin() (curses.window 的方法), 940
myrights() (imaplib.IMAP4 的方法), 1487

N

-N
    uuid 命令列選項, 1501
-n
    timeit 命令列選項, 1888
    uuid 命令列選項, 1501
    webbrowser 命令列選項, 1423
N_TOKENS (於 token 模組中), 2138
n_waiting (asyncio.Barrier 的屬性), 1110
n_waiting (threading.Barrier 的屬性), 981
NAK (於 curses.ascii 模組中), 956
--name
    uuid 命令列選項, 1501
Name (ast 中的類🄵), 2101
name (AttributeError 的屬性), 111
name (bz2.BZ2File 的屬性), 582
name (codecs.CodecInfo 的屬性), 194
name (compression.zstd.ZstdFile 的屬性), 565
name (contextvars.ContextVar 的屬性), 1068
name (ctypes.CField 的屬性), 852
name (doctest.DocTest 的屬性), 1739
name (email.headerregistry.BaseHeader 的屬性), 1286
name (enum.Enum 的屬性), 322
name (gzip.GzipFile 的屬性), 579
name (hashlib.hash 的屬性), 653
name (hmac.HMAC 的屬性), 663
name (http.cookiejar.Cookie 的屬性), 1528
name (ImportError 的屬性), 111
name (importlib.abc.FileLoader 的屬性), 2065
name (importlib.abc.Traversable 的屬性), 2067
name (importlib.machinery.AppleFrameworkLoader 的屬性), 2073
name (importlib.machinery.ExtensionFileLoader 的屬性), 2071
name (importlib.machinery.ModuleSpec 的屬性), 2072

name (*importlib.machinery.SourceFileLoader* 的屬性), 2070
name (*importlib.machinery.SourcelessFileLoader* 的屬性), 2071
name (*importlib.resources.abc.Traversable* 的屬性), 2082
name (*inspect.Parameter* 的屬性), 2024
name (*io.FileIO* 的屬性), 741
name (*logging.Logger* 的屬性), 761
name (*lzma.LZMAFile* 的屬性), 587
name (*multiprocessing.Process* 的屬性), 990
name (*multiprocessing.shared_memory.SharedMemory* 的屬性), 1027
name (*NameError* 的屬性), 112
name (*os.DirEntry* 的屬性), 700
name (*pathlib.PurePath* 的屬性), 449
name (*pyclbr.Class* 的屬性), 2145
name (*pyclbr.Function* 的屬性), 2144
name (*sys.thread_info* 的屬性), 1944
name (*tarfile.TarInfo* 的屬性), 610
name (*tempfile.TemporaryDirectory* 的屬性), 485
name (*threading.Thread* 的屬性), 973
name (*traceback.FrameSummary* 的屬性), 2006
name (於 *os* 模組中), 668
NAME (於 *token* 模組中), 2134
name (*webbrowser.controller* 的屬性), 1425
name (*xml.dom.Attr* 的屬性), 1391
name (*xml.dom.DocumentType* 的屬性), 1389
name (*zipfile.Path* 的屬性), 597
name() (於 *unicodedata* 模組中), 177
name2codepoint (於 *html.entities* 模組中), 1364
Named Shared Memory（附名共享記憶體）, 1026
named tuple（附名元組）, **2237**
NAMED_FLAGS (*enum.EnumCheck* 的屬性), 330
NamedExpr (*ast* 中的類🄵), 2104
NamedTemporaryFile() (於 *tempfile* 模組中), 483
NamedTuple (*typing* 中的類🄵), 1698
namedtuple() (於 *collections* 模組中), 273
NameError, 112
namelist() (*zipfile.ZipFile* 的方法), 594
nameprep() (於 *encodings.idna* 模組中), 210
namer (*logging.handlers.BaseRotatingHandler* 的屬性), 792
namereplace
    error handler's name（錯誤處理器名稱）, 197
namereplace_errors() (於 *codecs* 模組中), 199
names() (於 *tkinter.font* 模組中), 1592
--namespace
    uuid 命令列選項, 1501
Namespace (*argparse* 中的類🄵), 876
Namespace (*multiprocessing.managers* 中的類🄵), 1007
namespace package（命名空間套件）, **2238**
namespace() (*imaplib.IMAP4* 的方法), 1487
Namespace() (*multiprocessing.managers.SyncManager* 的方法), 1006
NAMESPACE_DNS (於 *uuid* 模組中), 1500
NAMESPACE_OID (於 *uuid* 模組中), 1500
NAMESPACE_URL (於 *uuid* 模組中), 1500
NAMESPACE_X500 (於 *uuid* 模組中), 1500
NamespaceErr, 1392
NamespaceLoader (*importlib.machinery* 中的類🄵), 2072
namespaceURI (*xml.dom.Node* 的屬性), 1387
namespace（命名空間）, **2238**
nametofont() (於 *tkinter.font* 模組中), 1592
NaN, 16
nan (*sys.hash_info* 的屬性), 1933
nan (於 *cmath* 模組中), 352
nan (於 *math* 模組中), 348
nanj (於 *cmath* 模組中), 352
NannyNag, 2143
napms() (於 *curses* 模組中), 932
nargs (*optparse.Option* 的屬性), 912
native_id (*threading.Thread* 的屬性), 973
nb_workers (*compression.zstd.CompressionParameter* 的屬性), 571
nbytes (*memoryview* 的屬性), 87
ncurses_version (於 *curses* 模組中), 942
ND (*inspect.BufferFlags* 的屬性), 2034
ndiff() (於 *difflib* 模組中), 163
ndim (*memoryview* 的屬性), 88
ne() (於 *operator* 模組中), 435
needs_input (*bz2.BZ2Decompressor* 的屬性), 584
needs_input (*compression.zstd.ZstdDecompressor* 的屬性), 567
needs_input (*lzma.LZMADecompressor* 的屬性), 589
neg() (於 *operator* 模組中), 437
nested scope（巢狀作用域）, **2238**
nested_scopes (於 *__future__* 模組中), 2011
netmask (*ipaddress.IPv4Network* 的屬性), 1550
netmask (*ipaddress.IPv6Network* 的屬性), 1553
NetmaskValueError, 1556
netrc
    module, 646
netrc (*netrc* 中的類🄵), 646
NetrcParseError, 646
netscape (*http.cookiejar.CookiePolicy* 的屬性), 1526
network (*ipaddress.IPv4Interface* 的屬性), 1554
network (*ipaddress.IPv6Interface* 的屬性), 1555
network_address (*ipaddress.IPv4Network* 的屬性), 1550
network_address (*ipaddress.IPv6Network* 的屬性), 1553
Never (於 *typing* 模組中), 1680
NEVER_EQ (於 *test.support* 模組中), 1838
new() (於 *hashlib* 模組中), 652
new() (於 *hmac* 模組中), 662
new-style class（新式類🄵）, **2238**
new_child() (*collections.ChainMap* 的方法), 264
new_class() (於 *types* 模組中), 303
new_event_loop() (*asyncio.AbstractEventLoopPolicy* 的方法), 1160
new_event_loop() (於 *asyncio* 模組中), 1121
new_panel() (於 *curses.panel* 模組中), 958

NEWLINE (於 *token* 模組中), 2134
newlines (*io.TextIOBase* 的屬性), 743
newpad() (於 *curses* 模組中), 932
--new-tab
    webbrowser 命令列選項, 1423
NewType (*typing* 中的類🄵), 1699
newwin() (於 *curses* 模組中), 932
--new-window
    webbrowser 命令列選項, 1423
next (*pdb command*), 1874
next()
    built-in function, 21
next() (*tarfile.TarFile* 的方法), 607
next() (*tkinter.ttk.Treeview* 的方法), 1611
next_minus() (*decimal.Context* 的方法), 369
next_minus() (*decimal.Decimal* 的方法), 362
next_plus() (*decimal.Context* 的方法), 369
next_plus() (*decimal.Decimal* 的方法), 362
next_toward() (*decimal.Context* 的方法), 369
next_toward() (*decimal.Decimal* 的方法), 362
nextafter() (於 *math* 模組中), 344
nextfile() (於 *fileinput* 模組中), 927
nextkey() (*dbm.gnu.gdbm* 的方法), 529
nextSibling (*xml.dom.Node* 的屬性), 1387
ngettext() (*gettext.GNUTranslations* 的方法), 1565
ngettext() (*gettext.NullTranslations* 的方法), 1564
ngettext() (於 *gettext* 模組中), 1562
nice() (於 *os* 模組中), 720
NIL (於 *uuid* 模組中), 1500
nis
    module, 2221
NL (於 *curses.ascii* 模組中), 955
NL (於 *token* 模組中), 2134
nl() (於 *curses* 模組中), 932
nl_langinfo() (於 *locale* 模組中), 1570
nlargest() (於 *heapq* 模組中), 288
nlst() (*ftplib.FTP* 的方法), 1477
nntplib
    module, 2221
NO (於 *tkinter.messagebox* 模組中), 1597
no_animation() (於 *turtle* 模組中), 1654
no_cache() (*zoneinfo.ZoneInfo* 的類🄵方法), 253
NO_EVENTS (*monitoring event*), 1948
no_proxy, 1439
no_site (*sys.flags* 的屬性), 1926
no_tracing() (於 *test.support* 模組中), 1843
no_type_check() (於 *typing* 模組中), 1710
no_type_check_decorator() (於 *typing* 模組中), 1710
no_user_site (*sys.flags* 的屬性), 1926
NoBoundaryInMultipartDefect, 1285
nocbreak() (於 *curses* 模組中), 933
NoDataAllowedErr, 1392
node (*uuid.UUID* 的屬性), 1498
node() (於 *platform* 模組中), 805
NoDefault (於 *typing* 模組中), 1713
nodelay() (*curses.window* 的方法), 940
nodeName (*xml.dom.Node* 的屬性), 1387
NodeTransformer (*ast* 中的類🄵), 2127
nodeType (*xml.dom.Node* 的屬性), 1386
nodeValue (*xml.dom.Node* 的屬性), 1387
NodeVisitor (*ast* 中的類🄵), 2126
noecho() (於 *curses* 模組中), 933
--no-ensure-ascii
    json 命令列選項, 1329
NOEXPR (於 *locale* 模組中), 1572
NOFLAG (於 *re* 模組中), 147
--no-indent
    json 命令列選項, 1329
NoModificationAllowedErr, 1392
--nonaliased
    platform 命令列選項, 808
NonCallableMagicMock (*unittest.mock* 中的類🄵), 1805
NonCallableMock (*unittest.mock* 中的類🄵), 1786
None (🄵建變數), 35
NoneType (於 *types* 模組中), 305
None (🄵建物件), 37
nonl() (於 *curses* 模組中), 933
Nonlocal (*ast* 中的類🄵), 2123
nonmember() (於 *enum* 模組中), 333
noop() (*imaplib.IMAP4* 的方法), 1487
noop() (*poplib.POP3* 的方法), 1482
NoOptionError, 643
NOP (*opcode*), 2157
noqiflush() (於 *curses* 模組中), 933
noraw() (於 *curses* 模組中), 933
--no-report
    trace 命令列選項, 1891
NoReturn (於 *typing* 模組中), 1680
NORMAL (於 *tkinter.font* 模組中), 1591
NORMAL_PRIORITY_CLASS (於 *subprocess* 模組中), 1057
NormalDist (*statistics* 中的類🄵), 404
normalize() (*decimal.Context* 的方法), 369
normalize() (*decimal.Decimal* 的方法), 362
normalize() (於 *locale* 模組中), 1574
normalize() (於 *unicodedata* 模組中), 178
normalize() (*xml.dom.Node* 的方法), 1388
normalize_encoding() (於 *encodings* 模組中), 209
NORMALIZE_WHITESPACE (於 *doctest* 模組中), 1731
normalvariate() (於 *random* 模組中), 388
normcase() (於 *os.path* 模組中), 472
normpath() (於 *os.path* 模組中), 472
NoSectionError, 643
NoSuchMailboxError, 1346
not
    operator (運算子), 38
Not (*ast* 中的類🄵), 2102
not in
    operator (運算子), 38, 46
not_() (於 *operator* 模組中), 436
NOT_TAKEN (*opcode*), 2157
NotADirectoryError, 117
notationDecl() (*xml.sax.handler.DTDHandler* 的方法), 1405

NotationDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
notations (*xml.dom.DocumentType* 的屬性), 1389
NotConnected, 1468
Notebook (*tkinter.ttk* 中的類🄵), 1605
NotEmptyError, 1347
NotEq (*ast* 中的類🄵), 2103
NOTEQUAL (於 *token* 模組中), 2137
NotFoundErr, 1392
notify() (*asyncio.Condition* 的方法), 1108
notify() (*threading.Condition* 的方法), 977
notify_all() (*asyncio.Condition* 的方法), 1108
notify_all() (*threading.Condition* 的方法), 978
notimeout() (*curses.window* 的方法), 940
NotImplemented (🄵建變數), 35
NotImplementedError, 112
NotImplementedType (於 *types* 模組中), 306
NotIn (*ast* 中的類🄵), 2103
NotRequired (於 *typing* 模組中), 1685
NOTSET (於 *logging* 模組中), 766
NotShareableError, 1043
NotStandaloneHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
NotSupportedErr, 1392
NotSupportedError, 552
--no-type-comments
    ast 命令列選項, 2129
noutrefresh() (*curses.window* 的方法), 940
NOVEMBER (於 *calendar* 模組中), 261
now() (*datetime.datetime* 的類🄵方法), 225
npgettext() (*gettext.GNUTranslations* 的方法), 1565
npgettext() (*gettext.NullTranslations* 的方法), 1564
npgettext() (於 *gettext* 模組中), 1562
NSIG (於 *signal* 模組中), 1250
nsmallest() (於 *heapq* 模組中), 288
NTEventLogHandler (*logging.handlers* 中的類🄵), 799
ntohl() (於 *socket* 模組中), 1188
ntohs() (於 *socket* 模組中), 1188
ntransfercmd() (*ftplib.FTP* 的方法), 1477
NUL (於 *curses.ascii* 模組中), 955
nullcontext() (於 *contextlib* 模組中), 1983
NullHandler (*logging* 中的類🄵), 791
NullTranslations (*gettext* 中的類🄵), 1563
num_addresses (*ipaddress.IPv4Network* 的屬性), 1550
num_addresses (*ipaddress.IPv6Network* 的屬性), 1553
num_tickets (*ssl.SSLContext* 的屬性), 1225
--number
    timeit 命令列選項, 1888
Number (*numbers* 中的類🄵), 337
NUMBER (於 *token* 模組中), 2134
number_class() (*decimal.Context* 的方法), 369
number_class() (*decimal.Decimal* 的方法), 362
numbers
    module, 337
numerator (*fractions.Fraction* 的屬性), 382
numerator (*numbers.Rational* 的屬性), 338
numeric() (於 *unicodedata* 模組中), 177
numeric（數值）
    conversions（轉🄵）, 39
    literals（字面值）, 38
    object（物件）, 38
    type（型🄵）, operations on（操作於）, 39
numinput() (於 *turtle* 模組中), 1656

O

-O
    ast 命令列選項, 2129
    dis 命令列選項, 2152
-o
    compileall 命令列選項, 2148
    cProfile 命令列選項, 1879
    doctest 命令列選項, 1727
    pickletools 命令列選項, 2174
    zipapp 命令列選項, 1914
O_APPEND (於 *os* 模組中), 682
O_ASYNC (於 *os* 模組中), 682
O_BINARY (於 *os* 模組中), 682
O_CLOEXEC (於 *os* 模組中), 682
O_CREAT (於 *os* 模組中), 682
O_DIRECT (於 *os* 模組中), 682
O_DIRECTORY (於 *os* 模組中), 682
O_DSYNC (於 *os* 模組中), 682
O_EVTONLY (於 *os* 模組中), 682
O_EXCL (於 *os* 模組中), 682
O_EXLOCK (於 *os* 模組中), 682
O_FSYNC (於 *os* 模組中), 682
O_NDELAY (於 *os* 模組中), 682
O_NOATIME (於 *os* 模組中), 682
O_NOCTTY (於 *os* 模組中), 682
O_NOFOLLOW (於 *os* 模組中), 682
O_NOFOLLOW_ANY (於 *os* 模組中), 682
O_NOINHERIT (於 *os* 模組中), 682
O_NONBLOCK (於 *os* 模組中), 682
O_PATH (於 *os* 模組中), 682
O_RANDOM (於 *os* 模組中), 682
O_RDONLY (於 *os* 模組中), 682
O_RDWR (於 *os* 模組中), 682
O_RSYNC (於 *os* 模組中), 682
O_SEQUENTIAL (於 *os* 模組中), 682
O_SHLOCK (於 *os* 模組中), 682
O_SHORT_LIVED (於 *os* 模組中), 682
O_SYMLINK (於 *os* 模組中), 682
O_SYNC (於 *os* 模組中), 682
O_TEMPORARY (於 *os* 模組中), 682
O_TEXT (於 *os* 模組中), 682
O_TMPFILE (於 *os* 模組中), 682
O_TRUNC (於 *os* 模組中), 682
O_WRONLY (於 *os* 模組中), 682
obj (*AttributeError* 的屬性), 111
obj (*memoryview* 的屬性), 87
object (*UnicodeError* 的屬性), 115

object (🄵建類🄵), 21
objects（物件）
    comparing（比較）, 38
    flattening（攤平）, 505
    marshalling, 505
    persistent（持續）, 505
    pickling, 505
    serializing（序列化）, 505
object（物件）, **2238**
    Boolean（布林）, 38
    bytearray（位元組陣列）, 48, 66, 68
    bytes（位元組）, 66, 67
    code（程式碼）, 103, 524
    complex number（🄵數）, 38
    dictionary（字典）, 91
    floating-point（浮點數）, 38
    GenericAlias（泛型🄵名）, 97
    integer（整數）, 38
    io.StringIO, 53
    list（串列）, 48, 49
    mapping（對映）, 91
    memoryview（記憶體視圖）, 66
    method（方法）, 103
    numeric（數值）, 38
    range, 51
    sequence（序列）, 46
    set（集合）, 89
    socket, 1174
    string（字串）, 52
    traceback, 1924, 2000
    tuple（元組）, 48, 50
    type（型🄵）, 30
    Union（聯合）, 101
oct()
    built-in function, 21
octal（八進位）
    literals（字面值）, 38
octdigits (於 *string* 模組中), 123
OCTOBER (於 *calendar* 模組中), 261
offset (*ctypes.CField* 的屬性), 852
offset (*SyntaxError* 的屬性), 114
offset (*tarfile.TarInfo* 的屬性), 611
offset (*traceback.TracebackException* 的屬性), 2004
offset (*xml.parsers.expat.ExpatError* 的屬性), 1417
offset_data (*tarfile.TarInfo* 的屬性), 611
OK (於 *curses* 模組中), 942
OK (於 *tkinter.messagebox* 模組中), 1597
ok_command() (*tkinter.filedialog.LoadFileDialog* 的方法), 1595
ok_command() (*tkinter.filedialog.SaveFileDialog* 的方法), 1595
ok_event() (*tkinter.filedialog.FileDialog* 的方法), 1595
OKCANCEL (於 *tkinter.messagebox* 模組中), 1597
old_value (*contextvars.Token* 的屬性), 1069
OleDLL (*ctypes* 中的類🄵), 836
on_motion() (*tkinter.dnd.DndHandler* 的方法), 1598
on_release() (*tkinter.dnd.DndHandler* 的方法), 1598
onclick() (於 *turtle* 模組中), 1655
ondrag() (於 *turtle* 模組中), 1650
onecmd() (*cmd.Cmd* 的方法), 960
onkey() (於 *turtle* 模組中), 1654
onkeypress() (於 *turtle* 模組中), 1655
onkeyrelease() (於 *turtle* 模組中), 1654
onrelease() (於 *turtle* 模組中), 1649
onscreenclick() (於 *turtle* 模組中), 1655
ontimer() (於 *turtle* 模組中), 1655
OP (於 *token* 模組中), 2134
OP_ALL (於 *ssl* 模組中), 1210
OP_CIPHER_SERVER_PREFERENCE (於 *ssl* 模組中), 1211
OP_ENABLE_KTLS (於 *ssl* 模組中), 1211
OP_ENABLE_MIDDLEBOX_COMPAT (於 *ssl* 模組中), 1211
OP_IGNORE_UNEXPECTED_EOF (於 *ssl* 模組中), 1211
OP_LEGACY_SERVER_CONNECT (於 *ssl* 模組中), 1211
OP_NO_COMPRESSION (於 *ssl* 模組中), 1211
OP_NO_RENEGOTIATION (於 *ssl* 模組中), 1211
OP_NO_SSLv2 (於 *ssl* 模組中), 1210
OP_NO_SSLv3 (於 *ssl* 模組中), 1210
OP_NO_TICKET (於 *ssl* 模組中), 1211
OP_NO_TLSv1 (於 *ssl* 模組中), 1210
OP_NO_TLSv1_1 (於 *ssl* 模組中), 1210
OP_NO_TLSv1_2 (於 *ssl* 模組中), 1210
OP_NO_TLSv1_3 (於 *ssl* 模組中), 1210
OP_SINGLE_DH_USE (於 *ssl* 模組中), 1211
OP_SINGLE_ECDH_USE (於 *ssl* 模組中), 1211
Open (*tkinter.filedialog* 中的類🄵), 1594
open()
    built-in function, 22
open() (*imaplib.IMAP4* 的方法), 1487
open() (*importlib.abc.Traversable* 的方法), 2067
open() (*importlib.resources.abc.Traversable* 的方法), 2083
open() (*pathlib.Path* 的方法), 459
open() (*tarfile.TarFile* 的類🄵方法), 607
open() (*urllib.request.OpenerDirector* 的方法), 1442
open() (於 *bz2* 模組中), 581
open() (於 *codecs* 模組中), 195
open() (於 *compression.zstd* 模組中), 564
open() (於 *dbm* 模組中), 526
open() (於 *dbm.dumb* 模組中), 530
open() (於 *dbm.gnu* 模組中), 528
open() (於 *dbm.ndbm* 模組中), 530
open() (於 *dbm.sqlite3* 模組中), 527
open() (於 *gzip* 模組中), 578
open() (於 *io* 模組中), 735
open() (於 *lzma* 模組中), 586
open() (於 *os* 模組中), 681
open() (於 *shelve* 模組中), 521
open() (於 *tarfile* 模組中), 602
open() (於 *tokenize* 模組中), 2140
open() (於 *wave* 模組中), 1557
open() (於 *webbrowser* 模組中), 1424

open() (*webbrowser.controller* 的方法), 1425
open() (*zipfile.Path* 的方法), 597
open() (*zipfile.ZipFile* 的方法), 594
open_binary() (於 *importlib.resources* 模組中), 2080
open_code() (於 *io* 模組中), 735
open_connection() (於 *asyncio* 模組中), 1098
open_flags (於 *dbm.gnu* 模組中), 528
open_new() (於 *webbrowser* 模組中), 1424
open_new() (*webbrowser.controller* 的方法), 1426
open_new_tab() (於 *webbrowser* 模組中), 1424
open_new_tab() (*webbrowser.controller* 的方法), 1426
open_osfhandle() (於 *msvcrt* 模組中), 2178
open_resource() (*importlib.abc.ResourceReader* 的方法), 2067
open_resource() (*importlib.resources.abc.ResourceReader* 的方法), 2082
open_text() (於 *importlib.resources* 模組中), 2080
open_unix_connection() (於 *asyncio* 模組中), 1099
open_urlresource() (於 *test.support* 模組中), 1843
OpenerDirector (*urllib.request* 中的類F), 1439
OpenKey() (於 *winreg* 模組中), 2183
OpenKeyEx() (於 *winreg* 模組中), 2183
openlog() (於 *syslog* 模組中), 2210
openpty() (於 *os* 模組中), 683
openpty() (於 *pty* 模組中), 2201
OpenSSL
    (使用於 hashlib 模組中), 651
    (用於 ssl 模組), 1202
OPENSSL_VERSION (於 *ssl* 模組中), 1213
OPENSSL_VERSION_INFO (於 *ssl* 模組中), 1213
OPENSSL_VERSION_NUMBER (於 *ssl* 模組中), 1213
OperationalError, 552
operations on (操作於)
    dictionary (字典) type (型F), 91
    integer (整數) type (型F), 40
    list (串列) type (型F), 48
    mapping (對映) type (型F), 91
    numeric (數值) type (型F), 39
    sequence (序列) type (型F), 46, 48
operations (操作)
    bitwise (位元), 40
    Boolean (布林), 37
    masking (遮罩), 40
    shifting (移位), 40
operation (操作)
    concatenation (串接), 46
    repetition (重F), 46
    slice (切片), 46
    subscript (下標), 46
operator
    module, 435
operator (運算子)
    - (F號), 38
    % (百分號), 38
    & (和號), 40
    * (星號), 38
    **, 38
    + (加號), 38
    / (斜F), 38
    //, 38
    < (小於), 38
    <<, 40
    <=, 38
    !=, 38
    ==, 38
    > (大於), 38
    >=, 38
    >>, 40
    ^ (插入符號), 40
    | (垂直F), 40
    ~ (波浪號), 40
    and, 37, 38
    comparison (比較), 38
    in, 38, 46
    is, 38
    is not, 38
    not, 38
    not in, 38, 46
    or, 37, 38
opmap (於 *dis* 模組中), 2173
opname (於 *dis* 模組中), 2173
optim_args_from_interpreter_flags() (於 *test.support* 模組中), 1840
--optimize
    ast 命令列選項, 2129
optimize (*sys.flags* 的屬性), 1926
optimize() (於 *pickletools* 模組中), 2175
optimized scope (最佳化作用域), **2238**
OPTIMIZED_BYTECODE_SUFFIXES (於 *importlib.machinery* 模組中), 2068
--option
    doctest 命令列選項, 1727
Option (*optparse* 中的類F), 912
Optional (於 *typing* 模組中), 1682
OptionConflictError, 925
OptionError, 925
OptionGroup (*optparse* 中的類F), 906
OptionParser (*optparse* 中的類F), 909
options (*doctest.Example* 的屬性), 1739
Options (*ssl* 中的類F), 1211
options (*ssl.SSLContext* 的屬性), 1225
options() (*configparser.ConfigParser* 的方法), 640
OptionValueError, 925
optionxform() (*configparser.ConfigParser* 的方法), 642
optparse
    module, 898
or
    operator (運算子), 37, 38
Or (*ast* 中的類F), 2102
or_() (於 *operator* 模組中), 437
ord()
    built-in function, 24

ordered_attributes (*xml.parsers.expat.xmlparser* 的屬性), 1414
OrderedDict (*collections* 中的類🄵), 277
OrderedDict (*typing* 中的類🄵), 1715
orig_argv (於 *sys* 模組中), 1937
origin (*importlib.machinery.ModuleSpec* 的屬性), 2072
origin_req_host (*urllib.request.Request* 的屬性), 1441
origin_server (*wsgiref.handlers.BaseHandler* 的屬性), 1433
os
    module, 667
    module（模組）, 2196
os_environ (*wsgiref.handlers.BaseHandler* 的屬性), 1432
OSError, 112
os.path
    module, 468
ossaudiodev
    module, 2222
OUT_TO_DEFAULT (於 *msvcrt* 模組中), 2179
OUT_TO_MSGBOX (於 *msvcrt* 模組中), 2179
OUT_TO_STDERR (於 *msvcrt* 模組中), 2179
outfile
    json 命令列選項, 1328
--output
    pickletools 命令列選項, 2174
    zipapp 命令列選項, 1914
output (*subprocess.CalledProcessError* 的屬性), 1046
output (*subprocess.TimeoutExpired* 的屬性), 1046
output (*unittest.TestCase* 的屬性), 1760
output() (*http.cookies.BaseCookie* 的方法), 1519
output() (*http.cookies.Morsel* 的方法), 1520
output_charset (*email.charset.Charset* 的屬性), 1314
output_codec (*email.charset.Charset* 的屬性), 1314
output_difference() (*doctest.OutputChecker* 的方法), 1743
OutputChecker (*doctest* 中的類🄵), 1742
OutputString() (*http.cookies.Morsel* 的方法), 1521
OutsideDestinationError, 604
Overflow (*decimal* 中的類🄵), 373
OverflowError, 113
overlap() (*statistics.NormalDist* 的方法), 404
overlap_log (*compression.zstd.CompressionParameter* 的屬性), 571
overlaps() (*ipaddress.IPv4Network* 的方法), 1551
overlaps() (*ipaddress.IPv6Network* 的方法), 1553
overlay() (*curses.window* 的方法), 940
overload() (於 *typing* 模組中), 1709
override() (於 *typing* 模組中), 1710
overwrite() (*curses.window* 的方法), 940
owner() (*pathlib.Path* 的方法), 465

P

-P
    dis 命令列選項, 2152
-p
    compileall 命令列選項, 2148
    http.server 命令列選項, 1517
    pdb 命令列選項, 1869
    pickletools 命令列選項, 2174
    timeit 命令列選項, 1888
    unittest-discover 命令列選項, 1750
    zipapp 命令列選項, 1914
p (*pdb command*), 1874
P_ALL (於 *os* 模組中), 726
P_DETACH (於 *os* 模組中), 724
P_NOWAIT (於 *os* 模組中), 723
P_NOWAITO (於 *os* 模組中), 723
P_OVERLAY (於 *os* 模組中), 724
P_PGID (於 *os* 模組中), 726
P_PID (於 *os* 模組中), 726
P_PIDFD (於 *os* 模組中), 726
P_WAIT (於 *os* 模組中), 724
pack() (*mailbox.MH* 的方法), 1336
pack() (*struct.Struct* 的方法), 193
pack() (於 *struct* 模組中), 187
pack_into() (*struct.Struct* 的方法), 193
pack_into() (於 *struct* 模組中), 188
PackageMetadata (*importlib.metadata* 中的類🄵), 2086
PackageNotFoundError, 2084
PackagePath (*importlib.metadata* 中的類🄵), 2087
packages_distributions() (於 *importlib.metadata* 模組中), 2088
package（套件）, 2044, **2238**
packed (*ipaddress.IPv4Address* 的屬性), 1545
packed (*ipaddress.IPv6Address* 的屬性), 1547
packing（元件）, 1586
packing（打包）
    binary（二進位）data（資料）, 187
PAGER, 1720
pair_content() (於 *curses* 模組中), 933
pair_number() (於 *curses* 模組中), 933
pairwise() (於 *itertools* 模組中), 416
Parameter (*inspect* 中的類🄵), 2024
ParameterizedMIMEHeader (*email.headerregistry* 中的類🄵), 1289
parameters (*inspect.Signature* 的屬性), 2023
parameter（參數）, **2238**
params (*email.headerregistry.ParameterizedMIMEHeader* 的屬性), 1289
ParamSpec (*ast* 中的類🄵), 2120
ParamSpec (*typing* 中的類🄵), 1695
ParamSpecArgs (於 *typing* 模組中), 1696
ParamSpecKwargs (於 *typing* 模組中), 1696
paramstyle (於 *sqlite3* 模組中), 536
pardir (於 *os* 模組中), 732
parent (*importlib.machinery.ModuleSpec* 的屬性), 2072
parent (*logging.Logger* 的屬性), 761
parent (*pathlib.PurePath* 的屬性), 449
parent (*pyclbr.Class* 的屬性), 2145
parent (*pyclbr.Function* 的屬性), 2144

parent (*urllib.request.BaseHandler* 的屬性), 1443
parent() (*tkinter.ttk.Treeview* 的方法), 1611
parent_process() (於 *multiprocessing* 模組中), 996
parentNode (*xml.dom.Node* 的屬性), 1386
parents (*collections.ChainMap* 的屬性), 264
parents (*pathlib.PurePath* 的屬性), 448
paretovariate() (於 *random* 模組中), 388
parse() (*doctest.DocTestParser* 的方法), 1741
parse() (*email.parser.BytesParser* 的方法), 1273
parse() (*email.parser.Parser* 的方法), 1274
parse() (*string.Formatter* 的方法), 124
parse() (*urllib.robotparser.RobotFileParser* 的方法), 1462
parse() (於 *ast* 模組中), 2124
parse() (於 *xml.dom.minidom* 模組中), 1394
parse() (於 *xml.dom.pulldom* 模組中), 1399
parse() (於 *xml.etree.ElementTree* 模組中), 1374
parse() (於 *xml.sax* 模組中), 1400
parse() (*xml.etree.ElementTree.ElementTree* 的方法), 1380
Parse() (*xml.parsers.expat.xmlparser* 的方法), 1412
parse() (*xml.sax.xmlreader.XMLReader* 的方法), 1408
parse_and_bind() (於 *readline* 模組中), 181
parse_args() (*argparse.ArgumentParser* 的方法), 874
parse_args() (*optparse.OptionParser* 的方法), 916
PARSE_COLNAMES (於 *sqlite3* 模組中), 536
parse_config_h() (於 *sysconfig* 模組中), 1956
PARSE_DECLTYPES (於 *sqlite3* 模組中), 536
parse_headers() (於 *http.client* 模組中), 1467
parse_intermixed_args() (*argparse.ArgumentParser* 的方法), 884
parse_known_args() (*argparse.ArgumentParser* 的方法), 883
parse_known_intermixed_args() (*argparse.ArgumentParser* 的方法), 884
parse_qs() (於 *urllib.parse* 模組中), 1454
parse_qsl() (於 *urllib.parse* 模組中), 1455
parseaddr() (於 *email.utils* 模組中), 1316
parsebytes() (*email.parser.BytesParser* 的方法), 1273
parsedate() (於 *email.utils* 模組中), 1317
parsedate_to_datetime() (於 *email.utils* 模組中), 1317
parsedate_tz() (於 *email.utils* 模組中), 1317
ParseError (*xml.etree.ElementTree* 中的類F), 1384
ParseFile() (*xml.parsers.expat.xmlparser* 的方法), 1412
ParseFlags() (於 *imaplib* 模組中), 1484
Parser (*email.parser* 中的類F), 1274
parser (*pathlib.PurePath* 的屬性), 447
ParserCreate() (於 *xml.parsers.expat* 模組中), 1412
ParseResult (*urllib.parse* 中的類F), 1458
ParseResultBytes (*urllib.parse* 中的類F), 1459
parsestr() (*email.parser.Parser* 的方法), 1274
parseString() (於 *xml.dom.minidom* 模組中), 1394
parseString() (於 *xml.dom.pulldom* 模組中), 1399
parseString() (於 *xml.sax* 模組中), 1400
ParsingError, 644
parsing（剖析）
    URL（統一資源定位器）, 1452
partial (*asyncio.IncompleteReadError* 的屬性), 1118
partial() (*imaplib.IMAP4* 的方法), 1487
partial() (於 *functools* 模組中), 429
partialmethod (*functools* 中的類F), 430
parties (*asyncio.Barrier* 的屬性), 1110
parties (*threading.Barrier* 的屬性), 981
partition() (*bytearray* 的方法), 72
partition() (*bytes* 的方法), 72
partition() (*str* 的方法), 59
partitioned (*http.cookies.Morsel* 的屬性), 1520
parts (*pathlib.PurePath* 的屬性), 447
Pass (*ast* 中的類F), 2109
pass_() (*poplib.POP3* 的方法), 1481
Paste（貼上）, 1621
patch() (於 *test.support* 模組中), 1844
patch() (於 *unittest.mock* 模組中), 1795
patch.dict() (於 *unittest.mock* 模組中), 1799
patch.multiple() (於 *unittest.mock* 模組中), 1800
patch.object() (於 *unittest.mock* 模組中), 1798
patch.stopall() (於 *unittest.mock* 模組中), 1802
PATH, 467, 497, 716, 717, 722, 723, 732, 1048, 1423, 1907, 2044
path (*http.cookiejar.Cookie* 的屬性), 1528
path (*http.cookies.Morsel* 的屬性), 1520
path (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
path (*ImportError* 的屬性), 111
path (*importlib.abc.FileLoader* 的屬性), 2065
path (*importlib.machinery.AppleFrameworkLoader* 的屬性), 2073
path (*importlib.machinery.ExtensionFileLoader* 的屬性), 2071
path (*importlib.machinery.FileFinder* 的屬性), 2070
path (*importlib.machinery.SourceFileLoader* 的屬性), 2070
path (*importlib.machinery.SourcelessFileLoader* 的屬性), 2071
path (*os.DirEntry* 的屬性), 700
Path (*pathlib* 中的類F), 454
path (於 *sys* 模組中), 1937
Path (*zipfile* 中的類F), 597
path based finder（基於路徑的尋檢器）, **2239**
Path browser（路徑F覽器）, 1617
path entry finder（路徑項目尋檢器）, **2239**
path entry hook（路徑項目F）, **2239**
path entry（路徑項目）, **2239**
path() (於 *importlib.resources* 模組中), 2081
path-like object（類路徑物件）, **2239**
path_hook() (*importlib.machinery.FileFinder* 的類F方法), 2070
path_hooks (於 *sys* 模組中), 1937
path_importer_cache (於 *sys* 模組中), 1937
path_mtime() (*importlib.abc.SourceLoader* 的方法), 2066

path_return_ok() (*http.cookiejar.CookiePolicy* 的方法), 1526
path_stats() (*importlib.abc.SourceLoader* 的方法), 2065
path_stats() (*importlib.machinery.SourceFileLoader* 的方法), 2070
pathconf() (於 *os* 模組中), 697
pathconf_names (於 *os* 模組中), 697
PathEntryFinder (*importlib.abc* 中的類🄵), 2062
PATHEXT, 497
PathFinder (*importlib.machinery* 中的類🄵), 2069
PathInfo (*pathlib.types* 中的類🄵), 468
pathlib
    module, 443
pathlib.types
    module, 468
PathLike (*os* 中的類🄵), 670
pathname2url() (於 *urllib.request* 模組中), 1437
pathsep (於 *os* 模組中), 732
Path.stem (於 *zipfile* 模組中), 598
Path.suffix (於 *zipfile* 模組中), 598
Path.suffixes (於 *zipfile* 模組中), 598
path (路徑)
    configuration (設定) file (檔案), 2044
    module (模組) search (搜尋), 492, 1937, 2043
    operations (操作), 443, 468
--pattern
    unittest-discover 命令列選項, 1750
Pattern (*re* 中的類🄵), 152
pattern (*re.Pattern* 的屬性), 153
pattern (*re.PatternError* 的屬性), 151
Pattern (*typing* 中的類🄵), 1716
PatternError, 151
pause() (於 *signal* 模組中), 1251
pause_reading() (*asyncio.ReadTransport* 的方法), 1149
pause_writing() (*asyncio.BaseProtocol* 的方法), 1152
PAX_FORMAT (於 *tarfile* 模組中), 606
pax_headers (*tarfile.TarFile* 的屬性), 609
pax_headers (*tarfile.TarInfo* 的屬性), 611
pbkdf2_hmac() (於 *hashlib* 模組中), 655
pd() (於 *turtle* 模組中), 1642
pdb
    module, 1868
Pdb (*pdb* 中的類🄵), 1871
pdb 命令列選項
    -c, 1869
    --command, 1869
    -m, 1869
    -p, 1869
    --pid, 1869
.pdbrc
    file (檔案), 1872
Pdb (pdb 中的類🄵), 1868
pdf() (*statistics.NormalDist* 的方法), 404
peek() (*bz2.BZ2File* 的方法), 582
peek() (*compression.zstd.ZstdFile* 的方法), 565
peek() (*gzip.GzipFile* 的方法), 579
peek() (*io.BufferedReader* 的方法), 742
peek() (*lzma.LZMAFile* 的方法), 586
peek() (*weakref.finalize* 的方法), 299
PEM_cert_to_DER_cert() (於 *ssl* 模組中), 1207
pen() (於 *turtle* 模組中), 1642
pencolor() (於 *turtle* 模組中), 1643
pending (*ssl.MemoryBIO* 的屬性), 1234
pending() (*ssl.SSLSocket* 的方法), 1218
PendingDeprecationWarning, 117
pendown() (於 *turtle* 模組中), 1642
pensize() (於 *turtle* 模組中), 1642
penup() (於 *turtle* 模組中), 1642
PEP, **2239**
PERCENT (於 *token* 模組中), 2137
PERCENTEQUAL (於 *token* 模組中), 2137
perf_counter() (於 *time* 模組中), 751
perf_counter_ns() (於 *time* 模組中), 751
perm() (於 *math* 模組中), 342
PermissionError, 117
permutations() (於 *itertools* 模組中), 416
persistence (持續性), 505
persistent_id() (*pickle.Pickler* 的方法), 508
persistent_id (pickle 協定), 513
persistent_load() (*pickle.Unpickler* 的方法), 510
persistent_load (pickle 協定), 513
persistent (持續)
    objects (物件), 505
PF_CAN (於 *socket* 模組中), 1179
PF_DIVERT (於 *socket* 模組中), 1180
PF_PACKET (於 *socket* 模組中), 1180
PF_RDS (於 *socket* 模組中), 1181
pformat() (*pprint.PrettyPrinter* 的方法), 313
pformat() (於 *pprint* 模組中), 312
pgettext() (*gettext.GNUTranslations* 的方法), 1565
pgettext() (*gettext.NullTranslations* 的方法), 1564
pgettext() (於 *gettext* 模組中), 1562
PGO (於 *test.support* 模組中), 1838
phase() (於 *cmath* 模組中), 350
pi (於 *cmath* 模組中), 352
pi (於 *math* 模組中), 348
pi() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
pickle
    module, 505
    module (模組), 310, 520, 521, 524
pickle 命令列選項
    pickle_file, 520
pickle() (於 *copyreg* 模組中), 521
pickle_file
    pickle 命令列選項, 520
    pickletools 命令列選項, 2174
PickleBuffer (*pickle* 中的類🄵), 510
PickleError, 508
Pickler (*pickle* 中的類🄵), 508
pickletools
    module, 2174
pickletools 命令列選項
    -a, 2174

--annotate, 2174
--indentlevel, 2174
-l, 2174
-m, 2174
--memo, 2174
-o, 2174
--output, 2174
-p, 2174
pickle_file, 2174
--preamble, 2174
pickling
objects（物件）, 505
PicklingError, 508
--pid
pdb 命令列選項, 1869
pid (*asyncio.subprocess.Process* 的屬性), 1114
pid (*multiprocessing.Process* 的屬性), 990
pid (*subprocess.Popen* 的屬性), 1055
PIDFD_NONBLOCK (於 *os* 模組中), 720
pidfd_open() (於 *os* 模組中), 720
pidfd_send_signal() (於 *signal* 模組中), 1251
PIPE (於 *subprocess* 模組中), 1046
Pipe() (於 *multiprocessing* 模組中), 993
pipe() (於 *os* 模組中), 683
pipe2() (於 *os* 模組中), 683
PIPE_BUF (於 *select* 模組中), 1238
pipe_connection_lost() (*asyncio.SubprocessProtocol* 的方法), 1154
pipe_data_received() (*asyncio.SubprocessProtocol* 的方法), 1154
PIPE_MAX_SIZE (於 *test.support* 模組中), 1838
pipes
module, 2222
pkgutil
module, 2053
placeholder (*textwrap.TextWrapper* 的屬性), 176
Placeholder (於 *functools* 模組中), 429
platform
module, 804
platform (於 *sys* 模組中), 1937
platform 命令列選項
--nonaliased, 808
--terse, 808
platform() (於 *platform* 模組中), 805
platlibdir (於 *sys* 模組中), 1938
PlaySound() (於 *winsound* 模組中), 2188
plist
file（檔案）, 647
plistlib
module, 647
plock() (於 *os* 模組中), 720
PLUS (於 *token* 模組中), 2136
plus() (*decimal.Context* 的方法), 370
PLUSEQUAL (於 *token* 模組中), 2137
pm() (於 *pdb* 模組中), 1870
POINTER() (於 *ctypes* 模組中), 844
pointer() (於 *ctypes* 模組中), 844
polar() (於 *cmath* 模組中), 350
Policy (*email.policy* 中的類🄵), 1279
poll() (*multiprocessing.connection.Connection* 的方法), 998
poll() (*select.devpoll* 的方法), 1239
poll() (*select.epoll* 的方法), 1240
poll() (*select.poll* 的方法), 1241
poll() (*subprocess.Popen* 的方法), 1053
poll() (於 *select* 模組中), 1238
PollSelector (*selectors* 中的類🄵), 1246
poly() (於 *turtle* 模組中), 1650
Pool (*multiprocessing.pool* 中的類🄵), 1010
pop() (*array.array* 的方法), 295
pop() (*collections.deque* 的方法), 270
pop() (*dict* 的方法), 94
pop() (*frozenset* 的方法), 91
pop() (*mailbox.Mailbox* 的方法), 1332
pop() (*sequence* 的方法), 49
POP3
protocol（協定）, 1480
POP3 (*poplib* 中的類🄵), 1480
POP3_SSL (*poplib* 中的類🄵), 1480
pop_all() (*contextlib.ExitStack* 的方法), 1988
POP_BLOCK (*opcode*), 2172
POP_EXCEPT (*opcode*), 2161
POP_ITER (*opcode*), 2157
POP_JUMP_IF_FALSE (*opcode*), 2166
POP_JUMP_IF_NONE (*opcode*), 2166
POP_JUMP_IF_NOT_NONE (*opcode*), 2166
POP_JUMP_IF_TRUE (*opcode*), 2166
pop_source() (*shlex.shlex* 的方法), 2193
POP_TOP (*opcode*), 2157
Popen (*subprocess* 中的類🄵), 1048
popen() (於 *os* 模組), 1238
popen() (於 *os* 模組中), 720
popitem() (*collections.OrderedDict* 的方法), 277
popitem() (*dict* 的方法), 94
popitem() (*mailbox.Mailbox* 的方法), 1332
popleft() (*collections.deque* 的方法), 270
poplib
module, 1480
port
http.server 命令列選項, 1517
port (*http.cookiejar.Cookie* 的屬性), 1528
port_specified (*http.cookiejar.Cookie* 的屬性), 1529
portion（部分）, **2239**
pos (*json.JSONDecodeError* 的屬性), 1326
pos (*re.Match* 的屬性), 155
pos (*re.PatternError* 的屬性), 151
pos (*tomllib.TOMLDecodeError* 的屬性), 645
pos() (於 *operator* 模組中), 437
pos() (於 *turtle* 模組中), 1640
position (*xml.etree.ElementTree.ParseError* 的屬性), 1384
position() (於 *turtle* 模組中), 1640
positional argument（位置引數）, **2239**
Positions (*dis* 中的類🄵), 2157
positions (*inspect.FrameInfo* 的屬性), 2029
positions (*inspect.Traceback* 的屬性), 2030

Positions.col_offset (於 *dis* 模組中), 2157
Positions.end_col_offset (於 *dis* 模組中), 2157
Positions.end_lineno (於 *dis* 模組中), 2157
Positions.lineno (於 *dis* 模組中), 2157
POSIX
    I/O control (I/O 控制), 2199
    threads (執行緒), 1071
posix
    module, 2196
POSIX Shared Memory (POSIX 共享記憶體), 1026
POSIX_FADV_DONTNEED (於 *os* 模組中), 683
POSIX_FADV_NOREUSE (於 *os* 模組中), 683
POSIX_FADV_NORMAL (於 *os* 模組中), 683
POSIX_FADV_RANDOM (於 *os* 模組中), 683
POSIX_FADV_SEQUENTIAL (於 *os* 模組中), 683
POSIX_FADV_WILLNEED (於 *os* 模組中), 683
posix_fadvise() (於 *os* 模組中), 683
posix_fallocate() (於 *os* 模組中), 683
posix_openpt() (於 *os* 模組中), 684
posix_spawn() (於 *os* 模組中), 721
POSIX_SPAWN_CLOSE (於 *os* 模組中), 721
POSIX_SPAWN_CLOSEFROM (於 *os* 模組中), 721
POSIX_SPAWN_DUP2 (於 *os* 模組中), 721
POSIX_SPAWN_OPEN (於 *os* 模組中), 721
posix_spawnp() (於 *os* 模組中), 722
PosixPath (*pathlib* 中的類🄵), 454
post_handshake_auth (*ssl.SSLContext* 的屬性), 1225
post_mortem() (於 *pdb* 模組中), 1870
post_setup() (*venv.EnvBuilder* 的方法), 1910
postcmd() (*cmd.Cmd* 的方法), 960
postloop() (*cmd.Cmd* 的方法), 961
Pow (*ast* 中的類🄵), 2102
pow()
    built-in function, 24
pow() (於 *math* 模組中), 345
pow() (於 *operator* 模組中), 437
power() (*decimal.Context* 的方法), 370
pp (*pdb command*), 1874
pp() (於 *pprint* 模組中), 311
pprint
    module, 311
pprint() (*pprint.PrettyPrinter* 的方法), 313
pprint() (於 *pprint* 模組中), 311
prcal() (於 *calendar* 模組中), 260
pread() (於 *os* 模組中), 684
preadv() (於 *os* 模組中), 684
--preamble
    pickletools 命令列選項, 2174
preamble (*email.message.EmailMessage* 的屬性), 1271
preamble (*email.message.Message* 的屬性), 1308
prec (*decimal.Context* 的屬性), 366
precmd() (*cmd.Cmd* 的方法), 960
prefix (於 *sys* 模組中), 1938
prefix (*xml.dom.Attr* 的屬性), 1391
prefix (*xml.dom.Node* 的屬性), 1387
prefix (*zipimport.zipimporter* 的屬性), 2053
PREFIXES (於 *site* 模組中), 2045
prefixlen (*ipaddress.IPv4Network* 的屬性), 1550
prefixlen (*ipaddress.IPv6Network* 的屬性), 1553
preloop() (*cmd.Cmd* 的方法), 961
prepare() (*graphlib.TopologicalSorter* 的方法), 335
prepare() (*logging.handlers.QueueHandler* 的方法), 802
prepare() (*logging.handlers.QueueListener* 的方法), 803
prepare_class() (於 *types* 模組中), 304
prepare_input_source() (於 *xml.sax.saxutils* 模組中), 1407
prepare_main() (*concurrent.interpreters.Interpreter* 的方法), 1042
PrepareProtocol (*sqlite3* 中的類🄵), 552
PrettyPrinter (*pprint* 中的類🄵), 312
prev() (*tkinter.ttk.Treeview* 的方法), 1611
previousSibling (*xml.dom.Node* 的屬性), 1387
print()
    built-in function, 25
print() (*traceback.TracebackException* 的方法), 2004
print_call_graph() (於 *asyncio* 模組中), 1119
print_callees() (*pstats.Stats* 的方法), 1884
print_callers() (*pstats.Stats* 的方法), 1883
print_exc() (*timeit.Timer* 的方法), 1888
print_exc() (於 *traceback* 模組中), 2001
print_exception() (於 *traceback* 模組中), 2001
print_function (於 *__future__* 模組中), 2011
print_help() (*argparse.ArgumentParser* 的方法), 883
print_last() (於 *traceback* 模組中), 2001
print_list() (於 *traceback* 模組中), 2002
print_stack() (*asyncio.Task* 的方法), 1095
print_stack() (於 *traceback* 模組中), 2001
print_stats() (*profile.Profile* 的方法), 1881
print_stats() (*pstats.Stats* 的方法), 1883
print_tb() (於 *traceback* 模組中), 2001
print_usage() (*argparse.ArgumentParser* 的方法), 883
print_usage() (*optparse.OptionParser* 的方法), 918
print_version() (*optparse.OptionParser* 的方法), 908
print_warning() (於 *test.support* 模組中), 1841
printable (於 *string* 模組中), 124
printdir() (*zipfile.ZipFile* 的方法), 595
printf 風格格式化, 64, 80
PRIO_DARWIN_BG (於 *os* 模組中), 673
PRIO_DARWIN_NONUI (於 *os* 模組中), 673
PRIO_DARWIN_PROCESS (於 *os* 模組中), 673
PRIO_DARWIN_THREAD (於 *os* 模組中), 673
PRIO_PGRP (於 *os* 模組中), 672
PRIO_PROCESS (於 *os* 模組中), 672
PRIO_USER (於 *os* 模組中), 672
PriorityQueue (*asyncio* 中的類🄵), 1117
PriorityQueue (*queue* 中的類🄵), 1064
prlimit() (於 *resource* 模組中), 2206
prmonth() (*calendar.TextCalendar* 的方法), 257
prmonth() (於 *calendar* 模組中), 260
ProactorEventLoop (*asyncio* 中的類🄵), 1141

process
    group（群組）, 671, 672
    id, 672
    id of parent, 672
    killing, 720
    scheduling priority(排程優先權), 672, 675
    signalling（信號）, 720
--process
    timeit 命令列選項, 1888
Process (*multiprocessing* 中的類🄵), 989
process() (*logging.LoggerAdapter* 的方法), 773
process_cpu_count() (於 *os* 模組中), 731
process_exited() (*asyncio.SubprocessProtocol* 的方法), 1154
process_request() (*socketserver.BaseServer* 的方法), 1506
process_time() (於 *time* 模組中), 751
process_time_ns() (於 *time* 模組中), 751
process_tokens() (於 *tabnanny* 模組中), 2143
ProcessError, 992
processes, light-weight（行程, 輕量級）, 1071
ProcessingInstruction() (於 *xml.etree.ElementTree* 模組中), 1374
processingInstruction() (*xml.sax.handler.ContentHandler* 的方法), 1405
ProcessingInstructionHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
ProcessLookupError, 117
processor time（處理器時間）, 751, 756
processor() (於 *platform* 模組中), 805
ProcessPoolExecutor (*concurrent.futures* 中的類🄵), 1035
prod() (於 *math* 模組中), 346
product() (於 *itertools* 模組中), 417
profile
    module, 1880
Profile (*profile* 中的類🄵), 1880
profile function, 968, 1931, 1939
profiler, 1931, 1939
profiling, deterministic, 1878
ProgrammingError, 552
Progressbar (*tkinter.ttk* 中的類🄵), 1606
--prompt
    venv 命令列選項, 1906
prompt (*cmd.Cmd* 的屬性), 961
prompts, interpreter（提示、直譯器）, 1939
propagate (*logging.Logger* 的屬性), 761
property (🄵建類🄵), 25
property list（屬性清單）, 647
property() (於 *enum* 模組中), 333
property_declaration_handler (於 *xml.sax.handler* 模組中), 1403
property_dom_node (於 *xml.sax.handler* 模組中), 1403
property_lexical_handler (於 *xml.sax.handler* 模組中), 1403
property_xml_string (於 *xml.sax.handler* 模組中), 1403
property.deleter()
    built-in function, 26
property.getter()
    built-in function, 26
PropertyMock (*unittest.mock* 中的類🄵), 1787
property.setter()
    built-in function, 26
prot_c() (*ftplib.FTP_TLS* 的方法), 1479
prot_p() (*ftplib.FTP_TLS* 的方法), 1479
proto (*socket.socket* 的屬性), 1198
--protocol
    http.server 命令列選項, 1517
Protocol (*asyncio* 中的類🄵), 1151
protocol (*ssl.SSLContext* 的屬性), 1225
Protocol (*typing* 中的類🄵), 1700
PROTOCOL_SSLv3 (於 *ssl* 模組中), 1209
PROTOCOL_SSLv23 (於 *ssl* 模組中), 1209
PROTOCOL_TLS (於 *ssl* 模組中), 1209
PROTOCOL_TLS_CLIENT (於 *ssl* 模組中), 1209
PROTOCOL_TLS_SERVER (於 *ssl* 模組中), 1209
PROTOCOL_TLSv1 (於 *ssl* 模組中), 1209
PROTOCOL_TLSv1_1 (於 *ssl* 模組中), 1209
PROTOCOL_TLSv1_2 (於 *ssl* 模組中), 1210
protocol_version (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
PROTOCOL_VERSION (*imaplib.IMAP4* 的屬性), 1490
ProtocolError (*xmlrpc.client* 中的類🄵), 1535
protocol（協定）
    context management（情境管理）, 96
    copy（🄵🄵）, 512
    FTP, 1451, 1474
    HTTP, 1451, 1463, 1466, 1511
    IMAP4, 1483
    IMAP4_SSL, 1483
    IMAP4_stream, 1483
    iterator（🄵代器）, 45
    POP3, 1480
    SMTP, 1490
provisional API（暫行 API）, **2239**
provisional package（暫行套件）, **2240**
proxy() (於 *weakref* 模組中), 297
proxyauth() (*imaplib.IMAP4* 的方法), 1487
ProxyBasicAuthHandler (*urllib.request* 中的類🄵), 1440
ProxyDigestAuthHandler (*urllib.request* 中的類🄵), 1440
ProxyHandler (*urllib.request* 中的類🄵), 1439
ProxyType (於 *weakref* 模組中), 300
ProxyTypes (於 *weakref* 模組中), 300
pryear() (*calendar.TextCalendar* 的方法), 258
ps1 (於 *sys* 模組中), 1939
ps2 (於 *sys* 模組中), 1939
pstats
    module, 1881
pstdev() (於 *statistics* 模組中), 399
pthread_getcpuclockid() (於 *time* 模組中), 749

pthread_kill() (於 signal 模組中), 1251
pthread_sigmask() (於 signal 模組中), 1252
pthreads, 1071
pthreads (sys._emscripten_info 的屬性), 1923
ptsname() (於 os 模組中), 685
pty
    module, 2201
    module（模組）, 683
pu() (於 turtle 模組中), 1642
publicId (xml.dom.DocumentType 的屬性), 1388
PullDom (xml.dom.pulldom 中的類🄵), 1399
punctuation (於 string 模組中), 124
punctuation_chars (shlex.shlex 的屬性), 2195
PurePath (pathlib 中的類🄵), 445
PurePosixPath (pathlib 中的類🄵), 445
PureWindowsPath (pathlib 中的類🄵), 446
purge() (於 re 模組中), 151
Purpose.CLIENT_AUTH (於 ssl 模組中), 1213
Purpose.SERVER_AUTH (於 ssl 模組中), 1213
push() (code.InteractiveConsole 的方法), 2049
push() (contextlib.ExitStack 的方法), 1987
push_async_callback() (contextlib.AsyncExitStack 的方法), 1988
push_async_exit() (contextlib.AsyncExitStack 的方法), 1988
PUSH_EXC_INFO (opcode), 2161
PUSH_NULL (opcode), 2169
push_source() (shlex.shlex 的方法), 2193
push_token() (shlex.shlex 的方法), 2193
put() (asyncio.Queue 的方法), 1116
put() (multiprocessing.Queue 的方法), 994
put() (multiprocessing.SimpleQueue 的方法), 995
put() (queue.Queue 的方法), 1065
put() (queue.SimpleQueue 的方法), 1067
put_nowait() (asyncio.Queue 的方法), 1116
put_nowait() (multiprocessing.Queue 的方法), 994
put_nowait() (queue.Queue 的方法), 1065
put_nowait() (queue.SimpleQueue 的方法), 1067
putch() (於 msvcrt 模組中), 2178
putenv() (於 os 模組中), 673
putheader() (http.client.HTTPConnection 的方法), 1471
putp() (於 curses 模組中), 933
putrequest() (http.client.HTTPConnection 的方法), 1470
putwch() (於 msvcrt 模組中), 2178
putwin() (curses.window 的方法), 940
pvariance() (於 statistics 模組中), 400
pwd
    module, 2197
    module（模組）, 470
pwd() (ftplib.FTP 的方法), 1478
pwrite() (於 os 模組中), 685
pwritev() (於 os 模組中), 685
py_compile
    module, 2145
Py_DEBUG (於 test.support 模組中), 1838
py_object (ctypes 中的類🄵), 849
PY_RESUME (monitoring event), 1948
PY_RETURN (monitoring event), 1948
PY_START (monitoring event), 1948
PY_THROW (monitoring event), 1948
PY_UNWIND (monitoring event), 1948
PY_YIELD (monitoring event), 1948
pycache_prefix (於 sys 模組中), 1923
PyCF_ALLOW_TOP_LEVEL_AWAIT (於 ast 模組中), 2128
PyCF_ONLY_AST (於 ast 模組中), 2128
PyCF_OPTIMIZED_AST (於 ast 模組中), 2128
PyCF_TYPE_COMMENTS (於 ast 模組中), 2128
PycInvalidationMode (py_compile 中的類🄵), 2146
pyclbr
    module, 2144
PyCompileError, 2145
PyDLL (ctypes 中的類🄵), 836
pydoc
    module, 1720
pyexpat
    module（模組）, 1411
PYFUNCTYPE() (於 ctypes 模組中), 840
--python
    zipapp 命令列選項, 1914
Python 3000, **2240**
Python Editor（Python 編輯器）, 1617
Python Enhancement Proposals
    PEP 1, 2239
    PEP 8, 28
    PEP 205, 300
    PEP 227, 2011
    PEP 235, 2059
    PEP 236, 2012
    PEP 237, 66, 82
    PEP 238, 2011, 2233
    PEP 246, 552
    PEP 249, 531, 532, 535, 547, 552, 555, 561
    PEP 0249#threadsafety, 537
    PEP 255, 2011
    PEP 263, 2060, 2140
    PEP 273, 2051
    PEP 278, 2243
    PEP 282, 499, 778
    PEP 292, 132
    PEP 302, 32, 492, 1937, 2051, 2054, 2055, 2057, 2060, 20632065, 2236
    PEP 305, 619
    PEP 307, 506
    PEP 324, 1044
    PEP 328, 32, 2011, 2060
    PEP 338, 2059
    PEP 342, 283
    PEP 343, 1992, 2011, 2230
    PEP 362, 2027, 2228, 2239
    PEP 366, 2059, 2060
    PEP 370, 2046
    PEP 378, 128

PEP 380#use-of-stopiteration-to-return-values, 1949
PEP 383, 197, 1174
PEP 387, 117, 118
PEP 393, 203, 1936
PEP 405, 1905
PEP 411, 1932, 1941, 1942, 2240
PEP 412, 426
PEP 420, 2060, 2238, 2239
PEP 421, 1934
PEP 428, 444
PEP 434, 1628
PEP 442, 2015
PEP 443, 2234
PEP 451, 1937, 20582060
PEP 453, 1903
PEP 461, 82
PEP 468, 277
PEP 475, 24, 117, 682, 686, 689, 726, 740, 752, 1192, 11941197, 12381242, 1246, 1254
PEP 479, 114, 2011
PEP 483, 2234
PEP 484, 101, 1669, 1678, 1691, 1709, 2098, 2124, 2125, 2128, 2227, 2233, 2234, 2243
PEP 485, 344, 352
PEP 488, 1852, 2060, 2074, 2145
PEP 489, 2060, 2069, 2071, 2075
PEP 492, 285, 2033, 22282230
PEP 495, 251
PEP 498, 2232
PEP 506, 664
PEP 515, 128, 382
PEP 519, 2239
PEP 524, 733
PEP 525, 285, 1932, 1941, 2034, 2228
PEP 526, 1684, 1699, 1969, 1977, 2036, 2124, 2128, 2227, 2243
PEP 529, 693, 1930, 1943
PEP 538, 1575
PEP 540, 668, 1575
PEP 544, 1678, 1700
PEP 552, 2060, 2146
PEP 554, 1040
PEP 557, 1969
PEP 560, 304, 305
PEP 563, 2011, 2036
PEP 565, 117
PEP 566, 2086
PEP 567, 1068, 1123, 1124, 1145
PEP 574, 507, 518
PEP 578, 1855, 1919
PEP 584, 264, 272, 277, 298, 307, 670
PEP 585, 101, 281, 307, 17131720, 2234
PEP 586, 1683
PEP 591, 1685, 1710
PEP 593, 1687, 1711
PEP 594, 2215, 22192223
PEP 597, 735
PEP 604, 102
PEP 610, 2089
PEP 612, 1671, 1676, 1683, 1696, 1719
PEP 613, 1681
PEP 615, 251
PEP 626, 2155
PEP 632, 2220
PEP 644, 1203
PEP 646, 1694
PEP 647, 1689
PEP 649, 1714, 2011, 2035, 2036, 2227
PEP 649#the-stringizer-and-the-fake-globals-environ 2038
PEP 655, 1685, 1705
PEP 667, 20, 1870
PEP 673, 1681
PEP 675, 1679
PEP 681, 1708
PEP 682, 127
PEP 683, 2234
PEP 684, 1040, 1934
PEP 686, 669, 735
PEP 688, 285
PEP 688#current-options, 284, 1716
PEP 692, 1690
PEP 695, 1679, 1691, 1693, 1695, 1696, 1720
PEP 698, 1710
PEP 702, 1968
PEP 703, 966, 2233, 2234
PEP 705, 1685
PEP 706, 612
PEP 709, 20
PEP 734, 1034, 1040, 1934
PEP 742, 1688
PEP 749, 2011, 2035, 2036
PEP 750, 134
PEP 3101, 124
PEP 3105, 2011
PEP 3107, 2036
PEP 3112, 2011
PEP 3115, 304, 2123
PEP 3116, 2243
PEP 3118, 83, 507
PEP 3119, 286, 1994
PEP 3120, 2060
PEP 3134, 110
PEP 3141, 337, 1994
PEP 3147, 1852, 2058, 2060, 2074, 2145, 2146, 21482150
PEP 3148, 1038
PEP 3149, 1919
PEP 3151, 117, 1177, 1237, 2206
PEP 3154, 507
PEP 3155, 2240
PEP 3333, 14261430, 1432, 1433
PEP 3333#input-and-error-streams, 1433
PEP 3333#optional-platform-specific-file-handling, 1434

PEP 3333#the-start-response-callable, 1433
python_branch() (於 *platform* 模組中), 805
python_build() (於 *platform* 模組中), 805
python_compiler() (於 *platform* 模組中), 805
PYTHON_CONTEXT_AWARE_WARNINGS, 1926, 1969
PYTHON_CPU_COUNT, 731, 996
PYTHON_DOM, 1385
PYTHON_GIL, 1926, 2234
python_implementation() (於 *platform* 模組中), 805
python_is_optimized() (於 *test.support* 模組中), 1839
PYTHON_JIT, 1935
python_revision() (於 *platform* 模組中), 805
PYTHON_THREAD_INHERIT_CONTEXT, 1926
python_version() (於 *platform* 模組中), 805
python_version_tuple() (於 *platform* 模組中), 805
PYTHONASYNCIODEBUG, 1137, 1171, 1722
PYTHONBREAKPOINT, 9, 1922
PYTHONCASEOK, 33
PYTHONCOERCECLOCALE, 669
PYTHONDEVMODE, 1721
PYTHONDONTWRITEBYTECODE, 1923
PYTHONFAULTHANDLER, 1722, 1865
PythonFinalizationError, 113
PYTHONHOME, 1846, 2091, 2092
Pythonic（Python 風格的）, **2240**
PYTHONINTMAXSTRDIGITS, 106, 1934
PYTHONIOENCODING, 668, 1943
PYTHONLEGACYWINDOWSFSENCODING, 1943
PYTHONLEGACYWINDOWSSTDIO, 1943
PYTHONMALLOC, 1722
python--m-py_compile 命令列選項
  -, 2147
  <file>, 2147
  -q, 2147
  --quiet, 2147
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令列選項
  -h, 554
  --help, 554
  -v, 554
  --version, 554
PYTHONNOUSERSITE, 2045
PYTHONPATH, 1846, 1937, 2091
PYTHONPLATLIBDIR, 2091
PYTHONPYCACHEPREFIX, 1923
PYTHONSAFEPATH, 1937, 2225
PYTHONSTARTUP, 183, 1076, 1624, 1934, 2045
PYTHONTRACEMALLOC, 1893, 1898
PYTHONTZPATH, 255
PYTHONUNBUFFERED, 1943
PYTHONUSERBASE, 2045
PYTHONUSERSITE, 1846
PYTHONUTF8, 669, 1943
PYTHONWARNDEFAULTENCODING, 735
PYTHONWARNINGS, 1721, 1964
PyZipFile (*zipfile* 中的類🄵), 598

Q

-q
  compileall 命令列選項, 2147
  python--m-py_compile 命令列選項, 2147
qiflush() (於 *curses* 模組中), 933
QName (*xml.etree.ElementTree* 中的類🄵), 1381
qsize() (*asyncio.Queue* 的方法), 1116
qsize() (*multiprocessing.Queue* 的方法), 993
qsize() (*queue.Queue* 的方法), 1065
qsize() (*queue.SimpleQueue* 的方法), 1067
qualified name（限定名稱）, **2240**
quantiles() (*statistics.NormalDist* 的方法), 404
quantiles() (於 *statistics* 模組中), 401
quantize() (*decimal.Context* 的方法), 370
quantize() (*decimal.Decimal* 的方法), 362
QueryInfoKey() (於 *winreg* 模組中), 2183
QueryReflectionKey() (於 *winreg* 模組中), 2185
QueryValue() (於 *winreg* 模組中), 2183
QueryValueEx() (於 *winreg* 模組中), 2184
QUESTION (於 *tkinter.messagebox* 模組中), 1597
queue
  module, 1064
Queue (*asyncio* 中的類🄵), 1115
Queue (*concurrent.interpreters* 中的類🄵), 1043
Queue (*multiprocessing* 中的類🄵), 993
Queue (*queue* 中的類🄵), 1064
queue (*sched.scheduler* 的屬性), 1064
Queue() (*multiprocessing.managers.SyncManager* 的方法), 1006
QueueEmpty, 1117
QueueEmptyError, 1043
QueueFull, 1117
QueueFullError, 1043
QueueHandler (*logging.handlers* 中的類🄵), 802
QueueListener (*logging.handlers* 中的類🄵), 803
QueueShutDown, 1117
quick_ratio() (*difflib.SequenceMatcher* 的方法), 167
--quiet
  python--m-py_compile 命令列選項, 2147
quiet (*sys.flags* 的屬性), 1926
quit (*pdb command*), 1876
quit (🄵建變數), 36
quit() (*ftplib.FTP* 的方法), 1478
quit() (*poplib.POP3* 的方法), 1482
quit() (*smtplib.SMTP* 的方法), 1496
quit() (*tkinter.filedialog.FileDialog* 的方法), 1595
quitting (*bdb.Bdb attribute*), 1863
quopri
  module, 1358
quote() (於 *email.utils* 模組中), 1316
quote() (於 *shlex* 模組中), 2191
quote() (於 *urllib.parse* 模組中), 1459
QUOTE_ALL (於 *csv* 模組中), 622
quote_from_bytes() (於 *urllib.parse* 模組中), 1459

QUOTE_MINIMAL (於 *csv* 模組中), 622
QUOTE_NONE (於 *csv* 模組中), 623
QUOTE_NONNUMERIC (於 *csv* 模組中), 622
QUOTE_NOTNULL (於 *csv* 模組中), 623
quote_plus() (於 *urllib.parse* 模組中), 1459
QUOTE_STRINGS (於 *csv* 模組中), 623
quoteattr() (於 *xml.sax.saxutils* 模組中), 1407
quotechar (*csv.Dialect* 的屬性), 624
quoted-printable（可列印字元）
    encoding（編碼）, 1358
quotes (*shlex.shlex* 的屬性), 2194
quoting (*csv.Dialect* 的屬性), 624

R

-R
    trace 命令列選項, 1891
-r
    compileall 命令列選項, 2148
    idle 命令列選項, 1624
    timeit 命令列選項, 1888
    trace 命令列選項, 1891
R_OK (於 *os* 模組中), 691
radians() (於 *math* 模組中), 346
radians() (於 *turtle* 模組中), 1642
radix (*sys.float_info* 的屬性), 1928
radix() (*decimal.Context* 的方法), 370
radix() (*decimal.Decimal* 的方法), 363
RADIXCHAR (於 *locale* 模組中), 1572
raise
    statement（陳述式）, 109
Raise (*ast* 中的類F), 2108
RAISE (*monitoring event*), 1948
raise_on_defect (*email.policy.Policy* 的屬性), 1280
raise_signal() (於 *signal* 模組中), 1251
RAISE_VARARGS (*opcode*), 2168
raiseExceptions (於 *logging* 模組中), 778
RAND_add() (於 *ssl* 模組中), 1206
RAND_bytes() (於 *ssl* 模組中), 1206
RAND_status() (於 *ssl* 模組中), 1206
randbelow() (於 *secrets* 模組中), 664
randbits() (於 *secrets* 模組中), 664
randbytes() (*random.Random* 的方法), 389
randbytes() (於 *random* 模組中), 385
randint() (於 *random* 模組中), 386
random
    module, 384
Random (*random* 中的類F), 388
random 命令列選項
    -c, 392
    --choice, 392
    -f, 393
    --float, 393
    -h, 392
    --help, 392
    -i, 392
    --integer, 392
random() (*random.Random* 的方法), 389
random() (於 *random* 模組中), 387
randrange() (於 *random* 模組中), 386
range
    object（物件）, 51
range (F建類F), 51
RARROW (於 *token* 模組中), 2138
ratio() (*difflib.SequenceMatcher* 的方法), 167
Rational (*numbers* 中的類F), 338
raw (*io.BufferedIOBase* 的屬性), 739
raw() (*pickle.PickleBuffer* 的方法), 510
raw() (於 *curses* 模組中), 933
raw_data_manager (於 *email.contentmanager* 模組中), 1293
raw_decode() (*json.JSONDecoder* 的方法), 1325
raw_input() (*code.InteractiveConsole* 的方法), 2049
RawArray() (於 *multiprocessing.sharedctypes* 模組中), 1002
RawConfigParser (*configparser* 中的類F), 642
RawDescriptionHelpFormatter (*argparse* 中的類F), 859
RawIOBase (*io* 中的類F), 739
RawPen (*turtle* 中的類F), 1659
RawTextHelpFormatter (*argparse* 中的類F), 859
RawTurtle (*turtle* 中的類F), 1659
RawValue() (於 *multiprocessing.sharedctypes* 模組中), 1002
RBRACE (於 *token* 模組中), 2137
re
    module, 139
    module（模組）, 490
    模組, 54
re (*re.Match* 的屬性), 155
READ (*inspect.BufferFlags* 的屬性), 2035
read() (*asyncio.StreamReader* 的方法), 1100
read() (*codecs.StreamReader* 的方法), 202
read() (*configparser.ConfigParser* 的方法), 640
read() (*http.client.HTTPResponse* 的方法), 1471
read() (*imaplib.IMAP4* 的方法), 1487
read() (*io.BufferedIOBase* 的方法), 740
read() (*io.BufferedReader* 的方法), 742
read() (*io.RawIOBase* 的方法), 739
read() (*io.Reader* 的方法), 746
read() (*io.TextIOBase* 的方法), 744
read() (*mimetypes.MimeTypes* 的方法), 1351
read() (*mmap.mmap* 的方法), 1259
read() (*sqlite3.Blob* 的方法), 551
read() (*ssl.MemoryBIO* 的方法), 1234
read() (*ssl.SSLSocket* 的方法), 1215
read() (*urllib.robotparser.RobotFileParser* 的方法), 1462
read() (於 *os* 模組中), 686
read() (*zipfile.ZipFile* 的方法), 595
read1() (*bz2.BZ2File* 的方法), 582
read1() (*io.BufferedIOBase* 的方法), 740
read1() (*io.BufferedReader* 的方法), 742
read1() (*io.BytesIO* 的方法), 742
read_binary() (於 *importlib.resources* 模組中), 2080
read_byte() (*mmap.mmap* 的方法), 1259

read_bytes() (*importlib.abc.Traversable* 的方法), 2068
read_bytes() (*importlib.resources.abc.Traversable* 的方法), 2083
read_bytes() (*pathlib.Path* 的方法), 459
read_bytes() (*zipfile.Path* 的方法), 598
read_dict() (*configparser.ConfigParser* 的方法), 641
read_environ() (於 *wsgiref.handlers* 模組中), 1433
read_events() (*xml.etree.ElementTree.XMLPullParser* 的方法), 1383
read_file() (*configparser.ConfigParser* 的方法), 640
read_history_file() (於 *readline* 模組中), 181
read_init_file() (於 *readline* 模組中), 181
read_mime_types() (於 *mimetypes* 模組中), 1349
read_string() (*configparser.ConfigParser* 的方法), 640
read_text() (*importlib.abc.Traversable* 的方法), 2068
read_text() (*importlib.resources.abc.Traversable* 的方法), 2083
read_text() (*pathlib.Path* 的方法), 459
read_text() (於 *importlib.resources* 模組中), 2081
read_text() (*zipfile.Path* 的方法), 598
read_token() (*shlex.shlex* 的方法), 2193
read_windows_registry() (*mimetypes.MimeTypes* 的方法), 1351
READABLE (於 *_tkinter* 模組中), 1591
readable() (*bz2.BZ2File* 的方法), 582
readable() (*io.IOBase* 的方法), 738
readall() (*io.RawIOBase* 的方法), 739
readbuffer_encode() (於 *codecs* 模組中), 196
Reader (*io* 中的類🄵), 746
reader() (於 *csv* 模組中), 619
ReadError, 604
readexactly() (*asyncio.StreamReader* 的方法), 1100
readfp() (*mimetypes.MimeTypes* 的方法), 1351
readframes() (*wave.Wave_read* 的方法), 1558
readinto() (*bz2.BZ2File* 的方法), 582
readinto() (*http.client.HTTPResponse* 的方法), 1471
readinto() (*io.BufferedIOBase* 的方法), 740
readinto() (*io.RawIOBase* 的方法), 739
readinto() (於 *os* 模組中), 686
readinto1() (*io.BufferedIOBase* 的方法), 740
readinto1() (*io.BytesIO* 的方法), 742
readline
    module, 180
readline() (*asyncio.StreamReader* 的方法), 1100
readline() (*codecs.StreamReader* 的方法), 202
readline() (*imaplib.IMAP4* 的方法), 1487
readline() (*io.IOBase* 的方法), 738
readline() (*io.TextIOBase* 的方法), 744
readline() (*mmap.mmap* 的方法), 1259
readlines() (*codecs.StreamReader* 的方法), 202
readlines() (*io.IOBase* 的方法), 738
readlink() (*pathlib.Path* 的方法), 456
readlink() (於 *os* 模組中), 697
readmodule() (於 *pyclbr* 模組中), 2144
readmodule_ex() (於 *pyclbr* 模組中), 2144
readonly (*memoryview* 的屬性), 88
ReadOnly (於 *typing* 模組中), 1685
ReadTransport (*asyncio* 中的類🄵), 1148
readuntil() (*asyncio.StreamReader* 的方法), 1101
readv() (於 *os* 模組中), 688
ready() (*multiprocessing.pool.AsyncResult* 的方法), 1012
Real (*numbers* 中的類🄵), 337
real (*numbers.Complex* 的屬性), 337
real_max_memuse (於 *test.support* 模組中), 1838
real_quick_ratio() (*difflib.SequenceMatcher* 的方法), 167
realpath() (於 *os.path* 模組中), 472
REALTIME_PRIORITY_CLASS (於 *subprocess* 模組中), 1057
reap_children() (於 *test.support* 模組中), 1843
reap_threads() (於 *test.support.threading_helper* 模組中), 1848
reason (*http.client.HTTPResponse* 的屬性), 1472
reason (*ssl.SSLError* 的屬性), 1205
reason (*UnicodeError* 的屬性), 115
reason (*urllib.error.HTTPError* 的屬性), 1461
reason (*urllib.error.URLError* 的屬性), 1461
reattach() (*tkinter.ttk.Treeview* 的方法), 1611
recent() (*imaplib.IMAP4* 的方法), 1487
reconfigure() (*io.TextIOWrapper* 的方法), 745
record_original_stdout() (於 *test.support* 模組中), 1840
RECORDS (*inspect.BufferFlags* 的屬性), 2034
records (*unittest.TestCase* 的屬性), 1760
RECORDS_RO (*inspect.BufferFlags* 的屬性), 2034
rect() (於 *cmath* 模組中), 350
rectangle() (於 *curses.textpad* 模組中), 953
RecursionError, 113
recursive_repr() (於 *reprlib* 模組中), 317
recv() (*multiprocessing.connection.Connection* 的方法), 998
recv() (*socket.socket* 的方法), 1194
recv_bytes() (*multiprocessing.connection.Connection* 的方法), 998
recv_bytes_into() (*multiprocessing.connection.Connection* 的方法), 998
recv_fds() (於 *socket* 模組中), 1191
recv_into() (*socket.socket* 的方法), 1196
recvfrom() (*socket.socket* 的方法), 1194
recvfrom_into() (*socket.socket* 的方法), 1195
recvmsg() (*socket.socket* 的方法), 1194
recvmsg_into() (*socket.socket* 的方法), 1195
redirect_request() (*urllib.request.HTTPRedirectHandler* 的方法), 1444
redirect_stderr() (於 *contextlib* 模組中), 1985
redirect_stdout() (於 *contextlib* 模組中), 1984
redisplay() (於 *readline* 模組中), 181
redrawln() (*curses.window* 的方法), 940
redrawwin() (*curses.window* 的方法), 941
reduce() (於 *functools* 模組中), 430
reducer_override() (*pickle.Pickler* 的方法), 509

ref (*weakref* 中的類🄵), 297
refcount_test() (於 *test.support* 模組中), 1843
reference count（參照計數）, **2240**
ReferenceError, 113
ReferenceType (於 *weakref* 模組中), 300
refold_source (*email.policy.EmailPolicy* 的屬性), 1282
refresh() (*curses.window* 的方法), 941
REG_BINARY (於 *winreg* 模組中), 2187
REG_DWORD (於 *winreg* 模組中), 2187
REG_DWORD_BIG_ENDIAN (於 *winreg* 模組中), 2187
REG_DWORD_LITTLE_ENDIAN (於 *winreg* 模組中), 2187
REG_EXPAND_SZ (於 *winreg* 模組中), 2187
REG_FULL_RESOURCE_DESCRIPTOR (於 *winreg* 模組中), 2187
REG_LINK (於 *winreg* 模組中), 2187
REG_MULTI_SZ (於 *winreg* 模組中), 2187
REG_NONE (於 *winreg* 模組中), 2187
REG_QWORD (於 *winreg* 模組中), 2187
REG_QWORD_LITTLE_ENDIAN (於 *winreg* 模組中), 2187
REG_RESOURCE_LIST (於 *winreg* 模組中), 2187
REG_RESOURCE_REQUIREMENTS_LIST (於 *winreg* 模組中), 2187
REG_SZ (於 *winreg* 模組中), 2187
RegexFlag (*re* 中的類🄵), 146
register() (*abc.ABCMeta* 的方法), 1995
register() (*argparse.ArgumentParser* 的方法), 885
register() (*functools.singledispatch* 的方法), 431
register() (*multiprocessing.managers.BaseManager* 的方法), 1005
register() (*select.devpoll* 的方法), 1239
register() (*select.epoll* 的方法), 1240
register() (*selectors.BaseSelector* 的方法), 1245
register() (*select.poll* 的方法), 1241
register() (於 *atexit* 模組中), 1999
register() (於 *codecs* 模組中), 195
register() (於 *faulthandler* 模組中), 1867
register() (於 *webbrowser* 模組中), 1424
register_adapter() (於 *sqlite3* 模組中), 535
register_archive_format() (於 *shutil* 模組中), 500
register_at_fork() (於 *os* 模組中), 722
register_callback() (於 *sys.monitoring* 模組中), 1950
register_converter() (於 *sqlite3* 模組中), 535
register_defect() (*email.policy.Policy* 的方法), 1280
register_dialect() (於 *csv* 模組中), 620
register_error() (於 *codecs* 模組中), 198
register_function() (*xmlrpc.server.CGIXMLRPCRequestHandler* 的方法), 1541
register_function() (*xmlrpc.server.SimpleXMLRPCServer* 的方法), 1538
register_instance() (*xmlrpc.server.CGIXMLRPCRequestHandler* 的方法), 1541
register_instance() (*xmlrpc.server.SimpleXMLRPCServer* 的方法), 1538
register_introspection_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* 的方法), 1542
register_introspection_functions() (*xmlrpc.server.SimpleXMLRPCServer* 的方法), 1539
register_multicall_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* 的方法), 1542
register_multicall_functions() (*xmlrpc.server.SimpleXMLRPCServer* 的方法), 1539
register_namespace() (於 *xml.etree.ElementTree* 模組中), 1374
register_optionflag() (於 *doctest* 模組中), 1732
register_shape() (於 *turtle* 模組中), 1657
register_unpack_format() (於 *shutil* 模組中), 501
registerDOMImplementation() (於 *xml.dom* 模組中), 1385
registerResult() (於 *unittest* 模組中), 1776
REGTYPE (於 *tarfile* 模組中), 605
regular package（正規套件）, **2240**
relative_to() (*pathlib.PurePath* 的方法), 452
relative（相對）
    URL（統一資源定位器）, 1452
release() (*_thread.lock* 的方法), 1073
release() (*asyncio.Condition* 的方法), 1108
release() (*asyncio.Lock* 的方法), 1106
release() (*asyncio.Semaphore* 的方法), 1109
release() (*logging.Handler* 的方法), 766
release() (*memoryview* 的方法), 85
release() (*multiprocessing.Lock* 的方法), 1000
release() (*multiprocessing.RLock* 的方法), 1001
release() (*pickle.PickleBuffer* 的方法), 510
release() (*threading.Condition* 的方法), 977
release() (*threading.Lock* 的方法), 974
release() (*threading.RLock* 的方法), 975
release() (*threading.Semaphore* 的方法), 978
release() (於 *platform* 模組中), 805
reload() (於 *importlib* 模組中), 2061
reload_environ() (於 *os* 模組中), 670
relpath() (於 *os.path* 模組中), 473
remainder() (*decimal.Context* 的方法), 370
remainder() (於 *math* 模組中), 343
remainder_near() (*decimal.Context* 的方法), 370
remainder_near() (*decimal.Decimal* 的方法), 363
remote_exec() (於 *sys* 模組中), 1942
RemoteDisconnected, 1468
remove() (*array.array* 的方法), 295
remove() (*collections.deque* 的方法), 270
remove() (*frozenset* 的方法), 91
remove() (*mailbox.Mailbox* 的方法), 1330
remove() (*mailbox.MH* 的方法), 1337

remove() (*sequence* 的方法), 49
remove() (於 *os* 模組中), 698
remove() (*xml.etree.ElementTree.Element* 的方法), 1378
remove_done_callback() (*asyncio.Future* 的方法), 1145
remove_done_callback() (*asyncio.Task* 的方法), 1095
remove_flag() (*mailbox.Maildir* 的方法), 1334
remove_flag() (*mailbox.MaildirMessage* 的方法), 1340
remove_flag() (*mailbox.mboxMessage* 的方法), 1342
remove_flag() (*mailbox.MMDFMessage* 的方法), 1346
remove_folder() (*mailbox.Maildir* 的方法), 1333
remove_folder() (*mailbox.MH* 的方法), 1336
remove_header() (*urllib.request.Request* 的方法), 1442
remove_history_item() (於 *readline* 模組中), 182
remove_label() (*mailbox.BabylMessage* 的方法), 1344
remove_option() (*configparser.ConfigParser* 的方法), 642
remove_option() (*optparse.OptionParser* 的方法), 917
remove_reader() (*asyncio.loop* 的方法), 1131
remove_section() (*configparser.ConfigParser* 的方法), 642
remove_sequence() (*mailbox.MHMessage* 的方法), 1343
remove_signal_handler() (*asyncio.loop* 的方法), 1134
remove_writer() (*asyncio.loop* 的方法), 1131
removeAttribute() (*xml.dom.Element* 的方法), 1390
removeAttributeNode() (*xml.dom.Element* 的方法), 1390
removeAttributeNS() (*xml.dom.Element* 的方法), 1390
removeChild() (*xml.dom.Node* 的方法), 1388
removedirs() (於 *os* 模組中), 698
removeFilter() (*logging.Handler* 的方法), 767
removeFilter() (*logging.Logger* 的方法), 765
removeHandler() (*logging.Logger* 的方法), 765
removeHandler() (於 *unittest* 模組中), 1776
removeprefix() (*bytearray* 的方法), 70
removeprefix() (*bytes* 的方法), 70
removeprefix() (*str* 的方法), 59
removeResult() (於 *unittest* 模組中), 1776
removesuffix() (*bytearray* 的方法), 70
removesuffix() (*bytes* 的方法), 70
removesuffix() (*str* 的方法), 59
removexattr() (於 *os* 模組中), 715
rename() (*ftplib.FTP* 的方法), 1477
rename() (*imaplib.IMAP4* 的方法), 1488
rename() (*pathlib.Path* 的方法), 464
rename() (於 *os* 模組中), 698
renames() (於 *os* 模組中), 699
reopenIfNeeded() (*logging.handlers.WatchedFileHandler* 的方法), 792
reorganize() (*dbm.gnu.gdbm* 的方法), 529
--repeat
    timeit 命令列選項, 1888
repeat() (*timeit.Timer* 的方法), 1887
repeat() (於 *itertools* 模組中), 417
repeat() (於 *timeit* 模組中), 1887
repetition (重🄵)
    operation (操作), 46
REPL, **2241**
replace
    error handler's name (錯誤處理器名稱), 197
replace() (*bytearray* 的方法), 72
replace() (*bytes* 的方法), 72
replace() (*curses.panel.Panel* 的方法), 959
replace() (*datetime.date* 的方法), 222
replace() (*datetime.datetime* 的方法), 230
replace() (*datetime.time* 的方法), 237
replace() (*inspect.Parameter* 的方法), 2025
replace() (*inspect.Signature* 的方法), 2023
replace() (*pathlib.Path* 的方法), 464
replace() (*str* 的方法), 59
replace() (*tarfile.TarInfo* 的方法), 611
replace() (於 *copy* 模組中), 310
replace() (於 *dataclasses* 模組中), 1975
replace() (於 *os* 模組中), 699
replace_errors() (於 *codecs* 模組中), 198
replace_header() (*email.message.EmailMessage* 的方法), 1267
replace_header() (*email.message.Message* 的方法), 1305
replace_history_item() (於 *readline* 模組中), 182
replace_whitespace (*textwrap.TextWrapper* 的屬性), 175
replaceChild() (*xml.dom.Node* 的方法), 1388
ReplacePackage() (於 *modulefinder* 模組中), 2056
--report
    trace 命令列選項, 1891
report() (*filecmp.dircmp* 的方法), 481
report() (*modulefinder.ModuleFinder* 的方法), 2056
REPORT_CDIFF (於 *doctest* 模組中), 1732
REPORT_ERRMODE (於 *msvcrt* 模組中), 2179
report_failure() (*doctest.DocTestRunner* 的方法), 1742
report_full_closure() (*filecmp.dircmp* 的方法), 481
REPORT_NDIFF (於 *doctest* 模組中), 1732
REPORT_ONLY_FIRST_FAILURE (於 *doctest* 模組中), 1732
report_partial_closure() (*filecmp.dircmp* 的方法), 481
report_start() (*doctest.DocTestRunner* 的方法), 1741
report_success() (*doctest.DocTestRunner* 的方法), 1741

REPORT_UDIFF (於 *doctest* 模組中), 1732
report_unexpected_exception() (*doctest.DocTestRunner* 的方法), 1742
REPORTING_FLAGS (於 *doctest* 模組中), 1732
Repr (*reprlib* 中的類🄵), 316
repr()
    built-in function, 26
repr() (*reprlib.Repr* 的方法), 318
repr() (於 *reprlib* 模組中), 316
repr1() (*reprlib.Repr* 的方法), 318
ReprEnum (*enum* 中的類🄵), 329
reprlib
    module, 316
request (*socketserver.BaseRequestHandler* 的屬性), 1507
Request (*urllib.request* 中的類🄵), 1438
request() (*http.client.HTTPConnection* 的方法), 1469
request_queue_size (*socketserver.BaseServer* 的屬性), 1506
request_rate() (*urllib.robotparser.RobotFileParser* 的方法), 1462
request_uri() (於 *wsgiref.util* 模組中), 1426
request_version (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
RequestHandlerClass (*socketserver.BaseServer* 的屬性), 1505
requestline (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
Required (於 *typing* 模組中), 1685
requires() (於 *importlib.metadata* 模組中), 2088
requires() (於 *test.support* 模組中), 1839
requires_bz2() (於 *test.support* 模組中), 1842
requires_docstrings() (於 *test.support* 模組中), 1842
requires_freebsd_version() (於 *test.support* 模組中), 1842
requires_gil_enabled() (於 *test.support* 模組中), 1842
requires_gzip() (於 *test.support* 模組中), 1842
requires_IEEE_754() (於 *test.support* 模組中), 1842
requires_limited_api() (於 *test.support* 模組中), 1842
requires_linux_version() (於 *test.support* 模組中), 1842
requires_lzma() (於 *test.support* 模組中), 1842
requires_mac_version() (於 *test.support* 模組中), 1842
requires_resource() (於 *test.support* 模組中), 1842
requires_zlib() (於 *test.support* 模組中), 1842
RERAISE (*monitoring event*), 1948
RERAISE (*opcode*), 2161
reschedule() (*asyncio.Timeout* 的方法), 1089
reserved (*zipfile.ZipInfo* 的屬性), 600
RESERVED_FUTURE (於 *uuid* 模組中), 1500
RESERVED_MICROSOFT (於 *uuid* 模組中), 1500
RESERVED_NCS (於 *uuid* 模組中), 1500
reset() (*asyncio.Barrier* 的方法), 1110
reset() (*bdb.Bdb* 的方法), 1861
reset() (*codecs.IncrementalDecoder* 的方法), 200
reset() (*codecs.IncrementalEncoder* 的方法), 200
reset() (*codecs.StreamReader* 的方法), 202
reset() (*codecs.StreamWriter* 的方法), 201
reset() (*contextvars.ContextVar* 的方法), 1068
reset() (*html.parser.HTMLParser* 的方法), 1361
reset() (*threading.Barrier* 的方法), 981
reset() (於 *turtle* 模組中), 1646
reset() (*xml.dom.pulldom.DOMEventStream* 的方法), 1400
reset() (*xml.sax.xmlreader.IncrementalParser* 的方法), 1409
reset_mock() (*unittest.mock.AsyncMock* 的方法), 1791
reset_mock() (*unittest.mock.Mock* 的方法), 1781
reset_peak() (於 *tracemalloc* 模組中), 1898
reset_prog_mode() (於 *curses* 模組中), 933
reset_shell_mode() (於 *curses* 模組中), 933
reset_tzpath() (於 *zoneinfo* 模組中), 255
resetbuffer() (*code.InteractiveConsole* 的方法), 2049
resetscreen() (於 *turtle* 模組中), 1653
resetty() (於 *curses* 模組中), 933
resetwarnings() (於 *warnings* 模組中), 1968
resize() (*bytearray* 的方法), 69
resize() (*curses.window* 的方法), 941
resize() (*mmap.mmap* 的方法), 1259
resize() (於 *ctypes* 模組中), 844
resize_term() (於 *curses* 模組中), 933
resizemode() (於 *turtle* 模組中), 1647
resizeterm() (於 *curses* 模組中), 933
resolution (*datetime.date* 的屬性), 221
resolution (*datetime.datetime* 的屬性), 228
resolution (*datetime.time* 的屬性), 236
resolution (*datetime.timedelta* 的屬性), 217
resolve() (*pathlib.Path* 的方法), 456
resolve_bases() (於 *types* 模組中), 304
resolve_name() (於 *importlib.util* 模組中), 2074
resolve_name() (於 *pkgutil* 模組中), 2055
resolveEntity() (*xml.sax.handler.EntityResolver* 的方法), 1405
resource
    module, 2206
resource_path() (*importlib.abc.ResourceReader* 的方法), 2067
resource_path() (*importlib.resources.abc.ResourceReader* 的方法), 2082
ResourceDenied, 1836
ResourceLoader (*importlib.abc* 中的類🄵), 2063
ResourceReader (*importlib.abc* 中的類🄵), 2066
ResourceReader (*importlib.resources.abc* 中的類🄵), 2082
ResourceWarning, 118
response() (*imaplib.IMAP4* 的方法), 1488
ResponseNotReady, 1468

responses (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
responses (於 *http.client* 模組中), 1468
restart (*pdb command*), 1876
restart_events() (*bdb.Bdb* 的方法), 1863
restart_events() (於 *sys.monitoring* 模組中), 1950
restore() (*test.support.SaveSignals* 的方法), 1845
restore() (於 *difflib* 模組中), 164
restype (*ctypes._CFuncPtr* 的屬性), 839
result() (*asyncio.Future* 的方法), 1144
result() (*asyncio.Task* 的方法), 1095
result() (*concurrent.futures.Future* 的方法), 1037
results() (*trace.Trace* 的方法), 1892
RESUME (*opcode*), 2170
resume_reading() (*asyncio.ReadTransport* 的方法), 1149
resume_writing() (*asyncio.BaseProtocol* 的方法), 1152
retr() (*poplib.POP3* 的方法), 1481
retrbinary() (*ftplib.FTP* 的方法), 1476
retrlines() (*ftplib.FTP* 的方法), 1476
RETRY (於 *tkinter.messagebox* 模組中), 1597
RETRYCANCEL (於 *tkinter.messagebox* 模組中), 1597
Return (*ast* 中的類🄵), 2122
return (*pdb command*), 1874
return_annotation (*inspect.Signature* 的屬性), 2023
RETURN_GENERATOR (*opcode*), 2171
return_ok() (*http.cookiejar.CookiePolicy* 的方法), 1526
RETURN_VALUE (*opcode*), 2160
return_value (*unittest.mock.Mock* 的屬性), 1783
returncode (*asyncio.subprocess.Process* 的屬性), 1114
returncode (*subprocess.CalledProcessError* 的屬性), 1046
returncode (*subprocess.CompletedProcess* 的屬性), 1045
returncode (*subprocess.Popen* 的屬性), 1055
retval (*pdb command*), 1877
reveal_type() (於 *typing* 模組中), 1706
reverse() (*array.array* 的方法), 296
reverse() (*collections.deque* 的方法), 270
reverse() (*sequence* 的方法), 49
reverse_order() (*pstats.Stats* 的方法), 1883
reverse_pointer (*ipaddress.IPv4Address* 的屬性), 1545
reverse_pointer (*ipaddress.IPv6Address* 的屬性), 1547
reversed()
    built-in function, 27
Reversible (*collections.abc* 中的類🄵), 283
Reversible (*typing* 中的類🄵), 1719
revert() (*http.cookiejar.FileCookieJar* 的方法), 1525
rewind() (*wave.Wave_read* 的方法), 1558
RFC
    RFC 821, 1490, 1492
    RFC 822, 754, 1294, 1311, 1471, 1493, 1495, 1496, 1564
    RFC 959, 1474
    RFC 1123, 754
    RFC 1321, 651
    RFC 1422, 1228, 1236
    RFC 1521, 1356, 1358
    RFC 1522, 1357, 1358
    RFC 1730, 1483
    RFC 1738, 1461
    RFC 1750, 1206
    RFC 1808, 1453, 1461
    RFC 1869, 1490, 1492
    RFC 1939, 1480
    RFC 2014, 662
    RFC 2045, 1263, 1267, 1289, 1305, 1306, 1311, 1352, 1355
    RFC 2045 Section 6.8, 1534
    RFC 2046, 1263, 1293, 1311
    RFC 2047, 1263, 1282, 1287, 1311, 1312, 1317
    RFC 2060, 1483, 1489
    RFC 2068, 1518
    RFC 2109, 15181523, 15271529
    RFC 2177, 1486
    RFC 2183, 1263, 1269, 1307
    RFC 2231, 1263, 1267, 1268, 13041306, 1311, 1318
    RFC 2295, 1465
    RFC 2324, 1464
    RFC 2342, 1487
    RFC 2368, 1461
    RFC 2373, 1545, 1546
    RFC 2396, 1455, 1459, 1461
    RFC 2397, 1447
    RFC 2449, 1481
    RFC 2518, 1464
    RFC 2595, 1480, 1482
    RFC 2616, 1427, 1430, 1444, 1445, 1461
    RFC 2616 Section 5.1.2, 1469
    RFC 2616 Section 14.23, 1469
    RFC 2640, 1474, 1475, 1479
    RFC 2732, 1461
    RFC 2774, 1465
    RFC 2821, 1263
    RFC 2822, 754, 1303, 1311, 1312, 1316, 1317, 1339, 1467, 1513
    RFC 2964, 1523
    RFC 2965, 1438, 1441, 1522, 1523, 15251528, 1530
    RFC 3056, 1548
    RFC 3171, 1545
    RFC 3229, 1464
    RFC 3280, 1216
    RFC 3330, 1546
    RFC 3454, 179
    RFC 3490, 208, 210
    RFC 3490 Section 3.1, 210
    RFC 3492, 208, 210
    RFC 3493, 1202
    RFC 3501, 1489

RFC 3542, 1189
RFC 3548, 1356, 1357
RFC 3659, 1477
RFC 3879, 1546, 1547
RFC 3927, 1546
RFC 3986, 1453, 1454, 1457, 1459, 1461, 1512
RFC 4007, 1547, 1548
RFC 4086, 1236
RFC 4122, 1497, 1499, 1500
RFC 4180, 619
RFC 4193, 1547
RFC 4217, 1478
RFC 4291, 1546, 1547
RFC 4380, 1548
RFC 4627, 1319, 1327
RFC 4648, 13521354, 1356, 2225
RFC 4918, 1464, 1465
RFC 4954, 1494
RFC 5161, 1485
RFC 5246, 1213, 1236
RFC 5280, 1204, 1206, 1236
RFC 5321, 1291
RFC 5322, 1263, 1264, 1273, 1276, 1277, 1279, 1281, 1282, 12851288, 1291, 1300, 1496
RFC 5424, 798
RFC 5735, 1546
RFC 5789, 1466
RFC 5842, 1464, 1465
RFC 5891, 210
RFC 5895, 210
RFC 5929, 1217
RFC 6066, 1212, 1222, 1236
RFC 6531, 1265, 1282, 1491
RFC 6532, 1263, 1264, 1273, 1282
RFC 6585, 1464, 1465
RFC 6855, 1485
RFC 6856, 1482
RFC 7159, 1319, 1326, 1327
RFC 7230, 1438, 1471
RFC 7301, 1212, 1221
RFC 7525, 1236
RFC 7693, 655
RFC 7725, 1464
RFC 7914, 655
RFC 8089, 455
RFC 8089 Section 3, 1437
RFC 8297, 1464
RFC 8305, 1126
RFC 8470, 1464
RFC 9110, 14641466
RFC 9562, 14971501
RFC 9562 Section 5.1, 1499
RFC 9562 Section 5.3, 1499
RFC 9562 Section 5.4, 1499
RFC 9562 Section 5.5, 1499
RFC 9562 Section 5.6, 1499
RFC 9562 Section 5.7, 1500
RFC 9562 Section 5.8, 1500
RFC 9562 Section 5.9, 1500
RFC 9562 Section 5.10, 1500
rfc2109 (*http.cookiejar.Cookie* 的屬性), 1529
rfc2109_as_netscape (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1527
rfc2965 (*http.cookiejar.CookiePolicy* 的屬性), 1526
RFC_4122 (於 *uuid* 模組中), 1500
rfile (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
rfile (*socketserver.DatagramRequestHandler* 的屬性), 1507
rfind() (*bytearray* 的方法), 72
rfind() (*bytes* 的方法), 72
rfind() (*mmap.mmap* 的方法), 1259
rfind() (*str* 的方法), 59
rgb_to_hls() (於 *colorsys* 模組中), 1560
rgb_to_hsv() (於 *colorsys* 模組中), 1560
rgb_to_yiq() (於 *colorsys* 模組中), 1560
rglob() (*pathlib.Path* 的方法), 461
right (*filecmp.dircmp* 的屬性), 482
right() (於 *turtle* 模組中), 1636
right_list (*filecmp.dircmp* 的屬性), 482
right_only (*filecmp.dircmp* 的屬性), 482
RIGHTSHIFT (於 *token* 模組中), 2137
RIGHTSHIFTEQUAL (於 *token* 模組中), 2138
rindex() (*bytearray* 的方法), 72
rindex() (*bytes* 的方法), 72
rindex() (*str* 的方法), 59
rjust() (*bytearray* 的方法), 74
rjust() (*bytes* 的方法), 74
rjust() (*str* 的方法), 59
rlcompleter
    module, 184
RLIM_INFINITY (於 *resource* 模組中), 2206
RLIMIT_AS (於 *resource* 模組中), 2207
RLIMIT_CORE (於 *resource* 模組中), 2207
RLIMIT_CPU (於 *resource* 模組中), 2207
RLIMIT_DATA (於 *resource* 模組中), 2207
RLIMIT_FSIZE (於 *resource* 模組中), 2207
RLIMIT_KQUEUES (於 *resource* 模組中), 2208
RLIMIT_MEMLOCK (於 *resource* 模組中), 2207
RLIMIT_MSGQUEUE (於 *resource* 模組中), 2207
RLIMIT_NICE (於 *resource* 模組中), 2207
RLIMIT_NOFILE (於 *resource* 模組中), 2207
RLIMIT_NPROC (於 *resource* 模組中), 2207
RLIMIT_NPTS (於 *resource* 模組中), 2208
RLIMIT_OFILE (於 *resource* 模組中), 2207
RLIMIT_RSS (於 *resource* 模組中), 2207
RLIMIT_RTPRIO (於 *resource* 模組中), 2208
RLIMIT_RTTIME (於 *resource* 模組中), 2208
RLIMIT_SBSIZE (於 *resource* 模組中), 2208
RLIMIT_SIGPENDING (於 *resource* 模組中), 2208
RLIMIT_STACK (於 *resource* 模組中), 2207
RLIMIT_SWAP (於 *resource* 模組中), 2208
RLIMIT_VMEM (於 *resource* 模組中), 2207
RLock (*multiprocessing* 中的類F), 1000
RLock (*threading* 中的類F), 975

RLock() (*multiprocessing.managers.SyncManager* 的方法), 1006
rmd() (*ftplib.FTP* 的方法), 1478
rmdir() (*pathlib.Path* 的方法), 465
rmdir() (於 *os* 模組中), 699
rmdir() (於 *test.support.os_helper* 模組中), 1850
rmtree() (於 *shutil* 模組中), 495
rmtree() (於 *test.support.os_helper* 模組中), 1850
RobotFileParser (*urllib.robotparser* 中的類🄵), 1462
robots.txt, 1462
rollback() (*sqlite3.Connection* 的方法), 538
rollover() (*tempfile.SpooledTemporaryFile* 的方法), 484
ROMAN (於 *tkinter.font* 模組中), 1591
--root
    ensurepip 命令列選項, 1904
root (*pathlib.PurePath* 的屬性), 448
rotate() (*collections.deque* 的方法), 270
rotate() (*decimal.Context* 的方法), 370
rotate() (*decimal.Decimal* 的方法), 363
rotate() (*logging.handlers.BaseRotatingHandler* 的方法), 793
RotatingFileHandler (*logging.handlers* 中的類🄵), 794
rotation_filename() (*logging.handlers.BaseRotatingHandler* 的方法), 793
rotator (*logging.handlers.BaseRotatingHandler* 的屬性), 793
round()
    built-in function, 27
ROUND_05UP (於 *decimal* 模組中), 372
ROUND_CEILING (於 *decimal* 模組中), 371
ROUND_DOWN (於 *decimal* 模組中), 371
ROUND_FLOOR (於 *decimal* 模組中), 371
ROUND_HALF_DOWN (於 *decimal* 模組中), 372
ROUND_HALF_EVEN (於 *decimal* 模組中), 372
ROUND_HALF_UP (於 *decimal* 模組中), 372
ROUND_UP (於 *decimal* 模組中), 372
Rounded (*decimal* 中的類🄵), 373
rounding (*decimal.Context* 的屬性), 366
rounds (*sys.float_info* 的屬性), 1928
Row (*sqlite3* 中的類🄵), 550
row_factory (*sqlite3.Connection* 的屬性), 547
row_factory (*sqlite3.Cursor* 的屬性), 550
rowcount (*sqlite3.Cursor* 的屬性), 550
RPAR (於 *token* 模組中), 2136
rpartition() (*bytearray* 的方法), 72
rpartition() (*bytes* 的方法), 72
rpartition() (*str* 的方法), 59
rpc_paths (*xmlrpc.server.SimpleXMLRPCRequestHandler* 的屬性), 1539
rpop() (*poplib.POP3* 的方法), 1481
RS (於 *curses.ascii* 模組中), 956
rset() (*poplib.POP3* 的方法), 1482
RShift (*ast* 中的類🄵), 2102
rshift() (於 *operator* 模組中), 437
rsplit() (*bytearray* 的方法), 74
rsplit() (*bytes* 的方法), 74
rsplit() (*str* 的方法), 60
RSQB (於 *token* 模組中), 2136
rstrip() (*bytearray* 的方法), 74
rstrip() (*bytes* 的方法), 74
rstrip() (*str* 的方法), 60
rt() (於 *turtle* 模組中), 1636
RTLD_DEEPBIND (於 *os* 模組中), 732
RTLD_GLOBAL (於 *os* 模組中), 732
RTLD_LAZY (於 *os* 模組中), 732
RTLD_LOCAL (於 *os* 模組中), 732
RTLD_NODELETE (於 *os* 模組中), 732
RTLD_NOLOAD (於 *os* 模組中), 732
RTLD_NOW (於 *os* 模組中), 732
ruler (*cmd.Cmd* 的屬性), 961
run (*pdb command*), 1876
Run script（執行🄵本）, 1619
run() (*asyncio.Runner* 的方法), 1077
run() (*bdb.Bdb* 的方法), 1864
run() (*contextvars.Context* 的方法), 1069
run() (*doctest.DocTestRunner* 的方法), 1742
run() (*multiprocessing.Process* 的方法), 989
run() (*pdb.Pdb* 的方法), 1871
run() (*profile.Profile* 的方法), 1881
run() (*sched.scheduler* 的方法), 1064
run() (*threading.Thread* 的方法), 972
run() (*trace.Trace* 的方法), 1892
run() (*unittest.IsolatedAsyncioTestCase* 的方法), 1766
run() (*unittest.TestCase* 的方法), 1756
run() (*unittest.TestSuite* 的方法), 1767
run() (*unittest.TextTestRunner* 的方法), 1773
run() (於 *asyncio* 模組中), 1076
run() (於 *pdb* 模組中), 1870
run() (於 *profile* 模組中), 1880
run() (於 *subprocess* 模組中), 1044
run() (*wsgiref.handlers.BaseHandler* 的方法), 1431
run_concurrently() (於 *test.support.threading_helper* 模組中), 1849
run_coroutine_threadsafe() (於 *asyncio* 模組中), 1092
run_docstring_examples() (於 *doctest* 模組中), 1735
run_forever() (*asyncio.loop* 的方法), 1122
run_in_executor() (*asyncio.loop* 的方法), 1135
run_in_subinterp() (於 *test.support* 模組中), 1844
run_module() (於 *runpy* 模組中), 2057
run_path() (於 *runpy* 模組中), 2058
run_python_until_end() (於 *test.support.script_helper* 模組中), 1846
run_script() (*modulefinder.ModuleFinder* 的方法), 2056
run_until_complete() (*asyncio.loop* 的方法), 1122
run_with_locale() (於 *test.support* 模組中), 1842
run_with_tz() (於 *test.support* 模組中), 1842
runcall() (*bdb.Bdb* 的方法), 1864
runcall() (*pdb.Pdb* 的方法), 1871

runcall() (*profile.Profile* 的方法), 1881
runcall() (於 *pdb* 模組中), 1870
runcode() (*code.InteractiveInterpreter* 的方法), 2048
runctx() (*bdb.Bdb* 的方法), 1864
runctx() (*profile.Profile* 的方法), 1881
runctx() (*trace.Trace* 的方法), 1892
runctx() (於 *profile* 模組中), 1880
runeval() (*bdb.Bdb* 的方法), 1864
runeval() (*pdb.Pdb* 的方法), 1871
runeval() (於 *pdb* 模組中), 1870
runfunc() (*trace.Trace* 的方法), 1892
Runner (*asyncio* 中的類🄵), 1077
running() (*concurrent.futures.Future* 的方法), 1037
runpy
    module, 2057
runsource() (*code.InteractiveInterpreter* 的方法), 2048
runtime (*sys._emscripten_info* 的屬性), 1923
runtime_checkable() (於 *typing* 模組中), 1701
RuntimeError, 113
RuntimeWarning, 118
RUSAGE_BOTH (於 *resource* 模組中), 2210
RUSAGE_CHILDREN (於 *resource* 模組中), 2209
RUSAGE_SELF (於 *resource* 模組中), 2209
RUSAGE_THREAD (於 *resource* 模組中), 2210
RWF_APPEND (於 *os* 模組中), 685
RWF_DSYNC (於 *os* 模組中), 685
RWF_HIPRI (於 *os* 模組中), 684
RWF_NOWAIT (於 *os* 模組中), 684
RWF_SYNC (於 *os* 模組中), 685

S

-S
    dis 命令列選項, 2152
-s
    calendar 命令列選項, 263
    compileall 命令列選項, 2148
    cProfile 命令列選項, 1879
    idle 命令列選項, 1624
    timeit 命令列選項, 1888
    trace 命令列選項, 1891
    unittest-discover 命令列選項, 1750
S (於 *re* 模組中), 147
S_ENFMT (於 *stat* 模組中), 478
S_IEXEC (於 *stat* 模組中), 479
S_IFBLK (於 *stat* 模組中), 477
S_IFCHR (於 *stat* 模組中), 477
S_IFDIR (於 *stat* 模組中), 477
S_IFDOOR (於 *stat* 模組中), 477
S_IFIFO (於 *stat* 模組中), 477
S_IFLNK (於 *stat* 模組中), 477
S_IFMT() (於 *stat* 模組中), 476
S_IFPORT (於 *stat* 模組中), 477
S_IFREG (於 *stat* 模組中), 477
S_IFSOCK (於 *stat* 模組中), 477
S_IFWHT (於 *stat* 模組中), 477
S_IMODE() (於 *stat* 模組中), 475
S_IREAD (於 *stat* 模組中), 478
S_IRGRP (於 *stat* 模組中), 478
S_IROTH (於 *stat* 模組中), 478
S_IRUSR (於 *stat* 模組中), 478
S_IRWXG (於 *stat* 模組中), 478
S_IRWXO (於 *stat* 模組中), 478
S_IRWXU (於 *stat* 模組中), 478
S_ISBLK() (於 *stat* 模組中), 475
S_ISCHR() (於 *stat* 模組中), 475
S_ISDIR() (於 *stat* 模組中), 475
S_ISDOOR() (於 *stat* 模組中), 475
S_ISFIFO() (於 *stat* 模組中), 475
S_ISGID (於 *stat* 模組中), 478
S_ISLNK() (於 *stat* 模組中), 475
S_ISPORT() (於 *stat* 模組中), 475
S_ISREG() (於 *stat* 模組中), 475
S_ISSOCK() (於 *stat* 模組中), 475
S_ISUID (於 *stat* 模組中), 478
S_ISVTX (於 *stat* 模組中), 478
S_ISWHT() (於 *stat* 模組中), 475
S_IWGRP (於 *stat* 模組中), 478
S_IWOTH (於 *stat* 模組中), 478
S_IWRITE (於 *stat* 模組中), 479
S_IWUSR (於 *stat* 模組中), 478
S_IXGRP (於 *stat* 模組中), 478
S_IXOTH (於 *stat* 模組中), 478
S_IXUSR (於 *stat* 模組中), 478
safe (*uuid.SafeUUID* 的屬性), 1497
safe_path (*sys.flags* 的屬性), 1926
safe_substitute() (*string.Template* 的方法), 132
saferepr() (於 *pprint* 模組中), 312
SafeUUID (*uuid* 中的類🄵), 1497
same_files (*filecmp.dircmp* 的屬性), 482
same_quantum() (*decimal.Context* 的方法), 370
same_quantum() (*decimal.Decimal* 的方法), 363
samefile() (*pathlib.Path* 的方法), 458
samefile() (於 *os.path* 模組中), 473
SameFileError, 493
sameopenfile() (於 *os.path* 模組中), 473
samesite (*http.cookies.Morsel* 的屬性), 1520
samestat() (於 *os.path* 模組中), 473
sample() (於 *random* 模組中), 387
samples() (*statistics.NormalDist* 的方法), 404
SATURDAY (於 *calendar* 模組中), 260
save() (*http.cookiejar.FileCookieJar* 的方法), 1524
save() (*test.support.SaveSignals* 的方法), 1845
save() (於 *turtle* 模組中), 1658
SaveAs (*tkinter.filedialog* 中的類🄵), 1594
SAVEDCWD (於 *test.support.os_helper* 模組中), 1849
SaveFileDialog (*tkinter.filedialog* 中的類🄵), 1595
SaveKey() (於 *winreg* 模組中), 2184
SaveSignals (*test.support* 中的類🄵), 1845
savetty() (於 *curses* 模組中), 933
SAX2DOM (*xml.dom.pulldom* 中的類🄵), 1399
SAXException, 1401
SAXNotRecognizedException, 1401
SAXNotSupportedException, 1401
SAXParseException, 1401
scaleb() (*decimal.Context* 的方法), 370

scaleb() (*decimal.Decimal* 的方法), 363
scandir() (於 *os* 模組中), 699
scanf（C 函式）, 156
sched
    module, 1063
SCHED_BATCH (於 *os* 模組中), 729
SCHED_DEADLINE (於 *os* 模組中), 729
SCHED_FIFO (於 *os* 模組中), 730
sched_get_priority_max() (於 *os* 模組中), 730
sched_get_priority_min() (於 *os* 模組中), 730
sched_getaffinity() (於 *os* 模組中), 730
sched_getparam() (於 *os* 模組中), 730
sched_getscheduler() (於 *os* 模組中), 730
SCHED_IDLE (於 *os* 模組中), 729
SCHED_NORMAL (於 *os* 模組中), 730
SCHED_OTHER (於 *os* 模組中), 729
sched_param (*os* 中的類🄵), 730
sched_priority (*os.sched_param* 的屬性), 730
SCHED_RESET_ON_FORK (於 *os* 模組中), 730
SCHED_RR (於 *os* 模組中), 730
sched_rr_get_interval() (於 *os* 模組中), 730
sched_setaffinity() (於 *os* 模組中), 730
sched_setparam() (於 *os* 模組中), 730
sched_setscheduler() (於 *os* 模組中), 730
SCHED_SPORADIC (於 *os* 模組中), 730
sched_yield() (於 *os* 模組中), 730
scheduler (*sched* 中的類🄵), 1063
SCM_CREDS2 (於 *socket* 模組中), 1183
scope_id (*ipaddress.IPv6Address* 的屬性), 1547
Screen (*turtle* 中的類🄵), 1659
screensize() (於 *turtle* 模組中), 1653
script_from_examples() (於 *doctest* 模組中), 1744
scroll() (*curses.window* 的方法), 941
ScrolledCanvas (*turtle* 中的類🄵), 1659
ScrolledText (*tkinter.scrolledtext* 中的類🄵), 1598
scrollok() (*curses.window* 的方法), 941
scrypt() (於 *hashlib* 模組中), 655
seal() (於 *unittest.mock* 模組中), 1814
search() (*imaplib.IMAP4* 的方法), 1488
search() (*re.Pattern* 的方法), 152
search() (於 *re* 模組中), 148
search_function() (於 *encodings* 模組中), 209
search_log (*compression.zstd.CompressionParameter* 的屬性), 570
search（搜尋）
    path（路徑）, module（模組）, 492, 1937, 2043
second (*datetime.datetime* 的屬性), 228
second (*datetime.time* 的屬性), 236
seconds (*datetime.timedelta* 的屬性), 217
secrets
    module, 663
SECTCRE (*configparser.ConfigParser* 的屬性), 636
sections() (*configparser.ConfigParser* 的方法), 640
secure (*http.cookiejar.Cookie* 的屬性), 1529
secure (*http.cookies.Morsel* 的屬性), 1520
Secure Sockets Layer（安全 socket 層）, 1202
security considerations（安全性注意事項）, 2223
security_level (*ssl.SSLContext* 的屬性), 1225
security（安全）
    http.server, 1518
see() (*tkinter.ttk.Treeview* 的方法), 1611
seed() (*random.Random* 的方法), 388
seed() (於 *random* 模組中), 385
seed_bits (*sys.hash_info* 的屬性), 1933
seek() (*io.IOBase* 的方法), 738
seek() (*io.TextIOBase* 的方法), 744
seek() (*io.TextIOWrapper* 的方法), 745
seek() (*mmap.mmap* 的方法), 1259
seek() (*sqlite3.Blob* 的方法), 551
SEEK_CUR (於 *os* 模組中), 681
SEEK_DATA (於 *os* 模組中), 681
SEEK_END (於 *os* 模組中), 681
SEEK_HOLE (於 *os* 模組中), 681
SEEK_SET (於 *os* 模組中), 681
seekable() (*bz2.BZ2File* 的方法), 582
seekable() (*io.IOBase* 的方法), 738
seekable() (*mmap.mmap* 的方法), 1259
select
    module, 1237
select() (*imaplib.IMAP4* 的方法), 1488
select() (*selectors.BaseSelector* 的方法), 1245
select() (*tkinter.ttk.Notebook* 的方法), 1605
select() (於 *select* 模組中), 1238
selected_alpn_protocol() (*ssl.SSLSocket* 的方法), 1217
selected_npn_protocol() (*ssl.SSLSocket* 的方法), 1217
selection() (*tkinter.ttk.Treeview* 的方法), 1611
selection_add() (*tkinter.ttk.Treeview* 的方法), 1612
selection_remove() (*tkinter.ttk.Treeview* 的方法), 1612
selection_set() (*tkinter.ttk.Treeview* 的方法), 1611
selection_toggle() (*tkinter.ttk.Treeview* 的方法), 1612
selector (*urllib.request.Request* 的屬性), 1441
SelectorEventLoop (*asyncio* 中的類🄵), 1140
SelectorKey (*selectors* 中的類🄵), 1244
selectors
    module, 1243
SelectSelector (*selectors* 中的類🄵), 1246
Self (於 *typing* 模組中), 1680
Semaphore (*asyncio* 中的類🄵), 1108
Semaphore (*multiprocessing* 中的類🄵), 1001
Semaphore (*threading* 中的類🄵), 978
Semaphore() (*multiprocessing.managers.SyncManager* 的方法), 1006
semaphores, binary（號🄵, 二進位）, 1071
SEMI (於 *token* 模組中), 2136
SEND (*opcode*), 2171
send() (*http.client.HTTPConnection* 的方法), 1471
send() (*imaplib.IMAP4* 的方法), 1488
send() (*logging.handlers.DatagramHandler* 的方法), 797
send() (*logging.handlers.SocketHandler* 的方法), 796

send() (*multiprocessing.connection.Connection* 的方法), 997
send() (*socket.socket* 的方法), 1196
send_bytes() (*multiprocessing.connection.Connection* 的方法), 998
send_error() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
send_fds() (於 *socket* 模組中), 1191
send_header() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
send_message() (*smtplib.SMTP* 的方法), 1496
send_response() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
send_response_only() (*http.server.BaseHTTPRequestHandler* 的方法), 1514
send_signal() (*asyncio.subprocess.Process* 的方法), 1113
send_signal() (*asyncio.SubprocessTransport* 的方法), 1151
send_signal() (*subprocess.Popen* 的方法), 1054
sendall() (*socket.socket* 的方法), 1196
sendcmd() (*ftplib.FTP* 的方法), 1476
sendfile() (*asyncio.loop* 的方法), 1130
sendfile() (*socket.socket* 的方法), 1197
sendfile() (於 *os* 模組中), 686
sendfile() (*wsgiref.handlers.BaseHandler* 的方法), 1433
SendfileNotAvailableError, 1118
sendmail() (*smtplib.SMTP* 的方法), 1495
sendmsg() (*socket.socket* 的方法), 1196
sendmsg_afalg() (*socket.socket* 的方法), 1197
sendto() (*asyncio.DatagramTransport* 的方法), 1150
sendto() (*socket.socket* 的方法), 1196
sentinel (*multiprocessing.Process* 的屬性), 991
sentinel (於 *unittest.mock* 模組中), 1807
sep (於 *os* 模組中), 732
SEPTEMBER (於 *calendar* 模組中), 261
Sequence (*collections.abc* 中的類F), 283
Sequence (*typing* 中的類F), 1717
SequenceMatcher (*difflib* 中的類F), 165
sequence（序列）, **2241**
  iteration（F代）, 45
  object（物件）, 46
  type（型F）, immutable（不可變）, 48
  type（型F）, mutable（可變）, 48
  type（型F）, operations on（操作於）, 46, 48
serialize() (*sqlite3.Connection* 的方法), 546
serializing（序列化）
  objects（物件）, 505
serve_forever() (*asyncio.Server* 的方法), 1140
serve_forever() (*socketserver.BaseServer* 的方法), 1505
Server (*asyncio* 中的類F), 1139
server (*http.server.BaseHTTPRequestHandler* 的屬性), 1512
server (*socketserver.BaseRequestHandler* 的屬性), 1507
server_activate() (*socketserver.BaseServer* 的方法), 1506
server_address (*socketserver.BaseServer* 的屬性), 1506
server_bind() (*socketserver.BaseServer* 的方法), 1507
server_close() (*socketserver.BaseServer* 的方法), 1505
server_hostname (*ssl.SSLSocket* 的屬性), 1218
server_side (*ssl.SSLSocket* 的屬性), 1218
server_software (*wsgiref.handlers.BaseHandler* 的屬性), 1432
server_version (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
server_version (*http.server.SimpleHTTPRequestHandler* 的屬性), 1515
ServerProxy (*xmlrpc.client* 中的類F), 1530
server（伺服器）
  WWW, 1511
service_actions() (*socketserver.BaseServer* 的方法), 1505
session (*ssl.SSLSocket* 的屬性), 1218
session_reused (*ssl.SSLSocket* 的屬性), 1218
session_stats() (*ssl.SSLContext* 的方法), 1224
Set (*ast* 中的類F), 2100
Set (*collections.abc* 中的類F), 284
Set (*typing* 中的類F), 1714
set (F建類F), 89
Set Breakpoint, 1621
set comprehension（集合綜合運算）, **2241**
set() (*asyncio.Event* 的方法), 1107
set() (*configparser.ConfigParser* 的方法), 641
set() (*configparser.RawConfigParser* 的方法), 643
set() (*contextvars.ContextVar* 的方法), 1068
set() (*http.cookies.Morsel* 的方法), 1520
set() (*multiprocessing.managers.SyncManager* 的方法), 1006
set() (*test.support.os_helper.EnvironmentVarGuard* 的方法), 1849
set() (*threading.Event* 的方法), 979
set() (*tkinter.ttk.Combobox* 的方法), 1603
set() (*tkinter.ttk.Spinbox* 的方法), 1603
set() (*tkinter.ttk.Treeview* 的方法), 1612
set() (*xml.etree.ElementTree.Element* 的方法), 1377
SET_ADD (*opcode*), 2160
set_allowed_domains() (*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
set_alpn_protocols() (*ssl.SSLContext* 的方法), 1221
set_app() (*wsgiref.simple_server.WSGIServer* 的方法), 1429
set_asyncgen_hooks() (於 *sys* 模組中), 1941
set_authorizer() (*sqlite3.Connection* 的方法), 542
set_auto_history() (於 *readline* 模組中), 182
set_blocked_domains()

(*http.cookiejar.DefaultCookiePolicy* 的方法), 1527
set_blocking() (於 *os* 模組中), 687
set_boundary() (*email.message.EmailMessage* 的方法), 1268
set_boundary() (*email.message.Message* 的方法), 1306
set_break() (*bdb.Bdb* 的方法), 1863
set_charset() (*email.message.Message* 的方法), 1303
set_children() (*tkinter.ttk.Treeview* 的方法), 1609
set_ciphers() (*ssl.SSLContext* 的方法), 1221
set_completer() (於 *readline* 模組中), 182
set_completer_delims() (於 *readline* 模組中), 183
set_completion_display_matches_hook() (於 *readline* 模組中), 183
set_content() (*email.contentmanager.ContentManager* 的方法), 1292
set_content() (*email.message.EmailMessage* 的方法), 1270
set_content() (於 *email.contentmanager* 模組中), 1293
set_continue() (*bdb.Bdb* 的方法), 1863
set_cookie() (*http.cookiejar.CookieJar* 的方法), 1524
set_cookie_if_ok() (*http.cookiejar.CookieJar* 的方法), 1524
set_coroutine_origin_tracking_depth() (於 *sys* 模組中), 1941
set_data() (*importlib.abc.SourceLoader* 的方法), 2066
set_data() (*importlib.machinery.SourceFileLoader* 的方法), 2070
set_date() (*mailbox.MaildirMessage* 的方法), 1340
set_debug() (*asyncio.loop* 的方法), 1137
set_debug() (於 *gc* 模組中), 2012
set_debuglevel() (*ftplib.FTP* 的方法), 1475
set_debuglevel() (*http.client.HTTPConnection* 的方法), 1470
set_debuglevel() (*poplib.POP3* 的方法), 1481
set_debuglevel() (*smtplib.SMTP* 的方法), 1492
set_default_backend() (於 *pdb* 模組中), 1870
set_default_executor() (*asyncio.loop* 的方法), 1136
set_default_type() (*email.message.EmailMessage* 的方法), 1268
set_default_type() (*email.message.Message* 的方法), 1305
set_default_verify_paths() (*ssl.SSLContext* 的方法), 1221
set_defaults() (*argparse.ArgumentParser* 的方法), 882
set_defaults() (*optparse.OptionParser* 的方法), 918
set_ecdh_curve() (*ssl.SSLContext* 的方法), 1223
set_errno() (於 *ctypes* 模組中), 844
set_error_mode() (於 *msvcrt* 模組中), 2178
set_escdelay() (於 *curses* 模組中), 934
set_event_loop() (*asyncio.AbstractEventLoopPolicy* 的方法), 1160
set_event_loop() (於 *asyncio* 模組中), 1121
set_event_loop_policy() (於 *asyncio* 模組中), 1160
set_events() (於 *sys.monitoring* 模組中), 1950
set_exception() (*asyncio.Future* 的方法), 1145
set_exception() (*concurrent.futures.Future* 的方法), 1038
set_exception_handler() (*asyncio.loop* 的方法), 1136
set_executable() (於 *multiprocessing* 模組中), 997
set_filter() (*tkinter.filedialog.FileDialog* 的方法), 1595
set_flags() (*mailbox.Maildir* 的方法), 1333
set_flags() (*mailbox.MaildirMessage* 的方法), 1340
set_flags() (*mailbox.mboxMessage* 的方法), 1342
set_flags() (*mailbox.MMDFMessage* 的方法), 1345
set_forkserver_preload() (於 *multiprocessing* 模組中), 997
set_from() (*mailbox.mboxMessage* 的方法), 1341
set_from() (*mailbox.MMDFMessage* 的方法), 1345
SET_FUNCTION_ATTRIBUTE (*opcode*), 2169
set_handle_inheritable() (於 *os* 模組中), 690
set_history_length() (於 *readline* 模組中), 181
set_info() (*mailbox.Maildir* 的方法), 1334
set_info() (*mailbox.MaildirMessage* 的方法), 1340
set_inheritable() (*socket.socket* 的方法), 1197
set_inheritable() (於 *os* 模組中), 690
set_int_max_str_digits() (於 *sys* 模組中), 1939
set_labels() (*mailbox.BabylMessage* 的方法), 1344
set_last_error() (於 *ctypes* 模組中), 844
set_local_events() (於 *sys.monitoring* 模組中), 1950
set_memlimit() (於 *test.support* 模組中), 1840
set_name() (*asyncio.Task* 的方法), 1096
set_next() (*bdb.Bdb* 的方法), 1862
set_nonstandard_attr() (*http.cookiejar.Cookie* 的方法), 1529
set_npn_protocols() (*ssl.SSLContext* 的方法), 1222
set_ok() (*http.cookiejar.CookiePolicy* 的方法), 1526
set_param() (*email.message.EmailMessage* 的方法), 1268
set_param() (*email.message.Message* 的方法), 1306
set_pasv() (*ftplib.FTP* 的方法), 1476
set_payload() (*email.message.Message* 的方法), 1303
set_pledged_input_size() (*compression.zstd.ZstdCompressor* 的方法), 566
set_policy() (*http.cookiejar.CookieJar* 的方法), 1524
set_pre_input_hook() (於 *readline* 模組中), 182
set_progress_handler() (*sqlite3.Connection* 的方法), 542
set_protocol() (*asyncio.BaseTransport* 的方法), 1149
set_proxy() (*urllib.request.Request* 的方法), 1442
set_psk_client_callback() (*ssl.SSLContext* 的方

法), 1226
set_psk_server_callback() (*ssl.SSLContext* 的方法), 1227
set_quit() (*bdb.Bdb* 的方法), 1863
set_result() (*asyncio.Future* 的方法), 1145
set_result() (*concurrent.futures.Future* 的方法), 1038
set_return() (*bdb.Bdb* 的方法), 1862
set_running_or_notify_cancel() (*concurrent.futures.Future* 的方法), 1037
set_selection() (*tkinter.filedialog.FileDialog* 的方法), 1595
set_seq1() (*difflib.SequenceMatcher* 的方法), 165
set_seq2() (*difflib.SequenceMatcher* 的方法), 165
set_seqs() (*difflib.SequenceMatcher* 的方法), 165
set_sequences() (*mailbox.MH* 的方法), 1336
set_sequences() (*mailbox.MHMessage* 的方法), 1343
set_server_documentation() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 的方法), 1543
set_server_documentation() (*xmlrpc.server.DocXMLRPCServer* 的方法), 1543
set_server_name() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 的方法), 1543
set_server_name() (*xmlrpc.server.DocXMLRPCServer* 的方法), 1542
set_server_title() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 的方法), 1543
set_server_title() (*xmlrpc.server.DocXMLRPCServer* 的方法), 1542
set_servername_callback() (*ssl.SSLContext* 的方法), 1222
set_start_method() (於 *multiprocessing* 模組中), 997
set_startup_hook() (於 *readline* 模組中), 182
set_step() (*bdb.Bdb* 的方法), 1862
set_subdir() (*mailbox.MaildirMessage* 的方法), 1340
set_tabsize() (於 *curses* 模組中), 934
set_task_factory() (*asyncio.loop* 的方法), 1125
set_threshold() (於 *gc* 模組中), 2013
set_trace() (*bdb.Bdb* 的方法), 1862
set_trace() (*pdb.Pdb* 的方法), 1871
set_trace() (於 *bdb* 模組中), 1864
set_trace() (於 *pdb* 模組中), 1870
set_trace_async() (於 *pdb* 模組中), 1870
set_trace_callback() (*sqlite3.Connection* 的方法), 542
set_tunnel() (*http.client.HTTPConnection* 的方法), 1470
set_type() (*email.message.Message* 的方法), 1306
set_unittest_reportflags() (於 *doctest* 模組中), 1737
set_unixfrom() (*email.message.EmailMessage* 的方法), 1265
set_unixfrom() (*email.message.Message* 的方法), 1302
set_until() (*bdb.Bdb* 的方法), 1862
SET_UPDATE (*opcode*), 2164
set_url() (*urllib.robotparser.RobotFileParser* 的方法), 1462
set_usage() (*optparse.OptionParser* 的方法), 918
set_userptr() (*curses.panel.Panel* 的方法), 959
set_visible() (*mailbox.BabylMessage* 的方法), 1344
set_wakeup_fd() (於 *signal* 模組中), 1252
set_write_buffer_limits() (*asyncio.WriteTransport* 的方法), 1150
setacl() (*imaplib.IMAP4* 的方法), 1488
setannotation() (*imaplib.IMAP4* 的方法), 1488
setattr()
    built-in function, 27
setAttribute() (*xml.dom.Element* 的方法), 1390
setAttributeNode() (*xml.dom.Element* 的方法), 1390
setAttributeNodeNS() (*xml.dom.Element* 的方法), 1390
setAttributeNS() (*xml.dom.Element* 的方法), 1390
SetBase() (*xml.parsers.expat.xmlparser* 的方法), 1413
setblocking() (*socket.socket* 的方法), 1197
setByteStream() (*xml.sax.xmlreader.InputSource* 的方法), 1410
setcbreak() (於 *tty* 模組中), 2201
setCharacterStream() (*xml.sax.xmlreader.InputSource* 的方法), 1410
SetComp (*ast* 中的類🄵), 2105
setcomptype() (*wave.Wave_write* 的方法), 1559
setconfig() (*sqlite3.Connection* 的方法), 546
setContentHandler() (*xml.sax.xmlreader.XMLReader* 的方法), 1408
setcontext() (於 *decimal* 模組中), 365
setDaemon() (*threading.Thread* 的方法), 973
setdefault() (*dict* 的方法), 94
setdefault() (*http.cookies.Morsel* 的方法), 1521
setdefaulttimeout() (於 *socket* 模組中), 1190
setdlopenflags() (於 *sys* 模組中), 1939
setDocumentLocator() (*xml.sax.handler.ContentHandler* 的方法), 1403
setDTDHandler() (*xml.sax.xmlreader.XMLReader* 的方法), 1409
setegid() (於 *os* 模組中), 673
setEncoding() (*xml.sax.xmlreader.InputSource* 的方法), 1410
setEntityResolver() (*xml.sax.xmlreader.XMLReader* 的方法), 1409
setErrorHandler() (*xml.sax.xmlreader.XMLReader*

的方法), 1409
seteuid() (於 os 模組中), 674
setFeature() (xml.sax.xmlreader.XMLReader 的方法), 1409
setfirstweekday() (calendar.Calendar 的方法), 256
setfirstweekday() (於 calendar 模組中), 259
setFormatter() (logging.Handler 的方法), 766
setframerate() (wave.Wave_write 的方法), 1559
setgid() (於 os 模組中), 674
setgroups() (於 os 模組中), 674
seth() (於 turtle 模組中), 1637
setheading() (於 turtle 模組中), 1637
sethostname() (於 socket 模組中), 1190
setinputsizes() (sqlite3.Cursor 的方法), 549
setitem() (於 operator 模組中), 438
setitimer() (於 signal 模組中), 1252
setLevel() (logging.Handler 的方法), 766
setLevel() (logging.Logger 的方法), 762
setlimit() (sqlite3.Connection 的方法), 545
setlocale() (於 locale 模組中), 1569
setLocale() (xml.sax.xmlreader.XMLReader 的方法), 1409
setLoggerClass() (於 logging 模組中), 777
setlogmask() (於 syslog 模組中), 2211
setLogRecordFactory() (於 logging 模組中), 777
setMaxConns() (urllib.request.CacheFTPHandler 的方法), 1447
setmode() (於 msvcrt 模組中), 2178
setName() (threading.Thread 的方法), 973
setnchannels() (wave.Wave_write 的方法), 1559
setnframes() (wave.Wave_write 的方法), 1559
setns() (於 os 模組中), 674
setoutputsize() (sqlite3.Cursor 的方法), 549
SetParamEntityParsing() (xml.parsers.expat.xmlparser 的方法), 1413
setparams() (wave.Wave_write 的方法), 1559
setpassword() (zipfile.ZipFile 的方法), 595
setpgid() (於 os 模組中), 675
setpgrp() (於 os 模組中), 674
setpos() (wave.Wave_read 的方法), 1558
setpos() (於 turtle 模組中), 1636
setposition() (於 turtle 模組中), 1636
setpriority() (於 os 模組中), 675
setprofile() (於 sys 模組中), 1939
setprofile() (於 threading 模組中), 968
setprofile_all_threads() (於 threading 模組中), 968
setProperty() (xml.sax.xmlreader.XMLReader 的方法), 1409
setPublicId() (xml.sax.xmlreader.InputSource 的方法), 1410
setquota() (imaplib.IMAP4 的方法), 1488
setraw() (於 tty 模組中), 2201
setrecursionlimit() (於 sys 模組中), 1940
setregid() (於 os 模組中), 675
SetReparseDeferralEnabled() (xml.parsers.expat.xmlparser 的方法), 1413
setresgid() (於 os 模組中), 675
setresuid() (於 os 模組中), 675
setreuid() (於 os 模組中), 675
setrlimit() (於 resource 模組中), 2206
setsampwidth() (wave.Wave_write 的方法), 1559
setscrreg() (curses.window 的方法), 941
setsid() (於 os 模組中), 675
setsockopt() (socket.socket 的方法), 1197
setstate() (codecs.IncrementalDecoder 的方法), 201
setstate() (codecs.IncrementalEncoder 的方法), 200
setstate() (random.Random 的方法), 388
setstate() (於 random 模組中), 385
setStream() (logging.StreamHandler 的方法), 791
setswitchinterval() (於 sys 模組中), 1940
setswitchinterval() (於 test.support 模組中), 1840
setSystemId() (xml.sax.xmlreader.InputSource 的方法), 1410
setsyx() (於 curses 模組中), 934
setTarget() (logging.handlers.MemoryHandler 的方法), 801
settimeout() (socket.socket 的方法), 1197
setTimeout() (urllib.request.CacheFTPHandler 的方法), 1447
settrace() (於 sys 模組中), 1940
settrace() (於 threading 模組中), 968
settrace_all_threads() (於 threading 模組中), 968
setuid() (於 os 模組中), 675
setundobuffer() (於 turtle 模組中), 1651
--setup
    timeit 命令列選項, 1888
setup() (socketserver.BaseRequestHandler 的方法), 1507
setUp() (unittest.TestCase 的方法), 1756
setup() (於 turtle 模組中), 1659
SETUP_ANNOTATIONS (opcode), 2161
SETUP_CLEANUP (opcode), 2172
setup_environ() (wsgiref.handlers.BaseHandler 的方法), 1432
SETUP_FINALLY (opcode), 2172
setup_python() (venv.EnvBuilder 的方法), 1909
setup_scripts() (venv.EnvBuilder 的方法), 1909
setup_testing_defaults() (於 wsgiref.util 模組中), 1427
SETUP_WITH (opcode), 2172
setUpClass() (unittest.TestCase 的方法), 1756
setUpModule() (於 unittest 模組中), 1775
setupterm() (於 curses 模組中), 934
SetValue() (於 winreg 模組中), 2184
SetValueEx() (於 winreg 模組中), 2184
setworldcoordinates() (於 turtle 模組中), 1653
setx() (於 turtle 模組中), 1637
setxattr() (於 os 模組中), 715
sety() (於 turtle 模組中), 1637
set (集合)

object（物件）, 89
SF_APPEND (於 *stat* 模組中), 480
SF_ARCHIVED (於 *stat* 模組中), 479
SF_DATALESS (於 *stat* 模組中), 480
SF_FIRMLINK (於 *stat* 模組中), 480
SF_IMMUTABLE (於 *stat* 模組中), 480
SF_MNOWAIT (於 *os* 模組中), 687
SF_NOCACHE (於 *os* 模組中), 687
SF_NODISKIO (於 *os* 模組中), 687
SF_NOUNLINK (於 *stat* 模組中), 480
SF_RESTRICTED (於 *stat* 模組中), 480
SF_SETTABLE (於 *stat* 模組中), 479
SF_SNAPSHOT (於 *stat* 模組中), 480
SF_SUPPORTED (於 *stat* 模組中), 479
SF_SYNC (於 *os* 模組中), 687
SF_SYNTHETIC (於 *stat* 模組中), 479
sha1() (於 *hashlib* 模組中), 652
sha3_224() (於 *hashlib* 模組中), 653
sha3_256() (於 *hashlib* 模組中), 653
sha3_384() (於 *hashlib* 模組中), 653
sha3_512() (於 *hashlib* 模組中), 653
sha224() (於 *hashlib* 模組中), 652
sha256() (於 *hashlib* 模組中), 652
sha384() (於 *hashlib* 模組中), 652
sha512() (於 *hashlib* 模組中), 652
shake_128() (於 *hashlib* 模組中), 654
shake_256() (於 *hashlib* 模組中), 654
shape (*memoryview* 的屬性), 88
Shape (*turtle* 中的類🄵), 1660
shape() (於 *turtle* 模組中), 1647
shapesize() (於 *turtle* 模組中), 1647
shapetransform() (於 *turtle* 模組中), 1648
share() (*socket.socket* 的方法), 1198
ShareableList (*multiprocessing.shared_memory* 中的類🄵), 1029
ShareableList() (*multiprocessing.managers.SharedMemoryManager* 的方法), 1028
Shared Memory（共享記憶體）, 1026
shared_ciphers() (*ssl.SSLSocket* 的方法), 1217
shared_memory (*sys._emscripten_info* 的屬性), 1923
SharedMemory (*multiprocessing.shared_memory* 中的類🄵), 1026
SharedMemory() (*multiprocessing.managers.SharedMemoryManager* 的方法), 1028
SharedMemoryManager (*multiprocessing.managers* 中的類🄵), 1028
shearfactor() (於 *turtle* 模組中), 1648
Shelf (*shelve* 中的類🄵), 522
shelve
    module, 521
    module（模組）, 524
shield() (於 *asyncio* 模組中), 1087
shift() (*decimal.Context* 的方法), 370
shift() (*decimal.Decimal* 的方法), 363
shift_path_info() (於 *wsgiref.util* 模組中), 1426
shifting（移位）
    operations（操作）, 40
shlex
    module, 2191
shlex (*shlex* 中的類🄵), 2192
shm (*multiprocessing.shared_memory.ShareableList* 的屬性), 1030
SHORT_TIMEOUT (於 *test.support* 模組中), 1837
shortDescription() (*unittest.TestCase* 的方法), 1764
shorten() (於 *textwrap* 模組中), 173
shouldFlush() (*logging.handlers.BufferingHandler* 的方法), 800
shouldFlush() (*logging.handlers.MemoryHandler* 的方法), 801
shouldRollover() (*logging.handlers.RotatingFileHandler* 的方法), 794
shouldRollover() (*logging.handlers.TimedRotatingFileHandler* 的方法), 795
shouldStop (*unittest.TestResult* 的屬性), 1770
show() (*curses.panel.Panel* 的方法), 959
show() (*tkinter.commondialog.Dialog* 的方法), 1595
show() (*tkinter.messagebox.Message* 的方法), 1596
show_code() (於 *dis* 模組中), 2153
show_flag_values() (於 *enum* 模組中), 333
--show-caches
    dis 命令列選項, 2152
--show-empty
    ast 命令列選項, 2129
showerror() (於 *tkinter.messagebox* 模組中), 1596
showinfo() (於 *tkinter.messagebox* 模組中), 1596
--show-offsets
    dis 命令列選項, 2152
--show-positions
    dis 命令列選項, 2152
showsyntaxerror() (*code.InteractiveInterpreter* 的方法), 2048
showtraceback() (*code.InteractiveInterpreter* 的方法), 2048
showturtle() (於 *turtle* 模組中), 1646
showwarning() (於 *tkinter.messagebox* 模組中), 1596
showwarning() (於 *warnings* 模組中), 1967
shuffle() (於 *random* 模組中), 386
SHUT_RD (於 *socket* 模組中), 1184
SHUT_RDWR (於 *socket* 模組中), 1184
SHUT_WR (於 *socket* 模組中), 1184
ShutDown, 1065
shutdown() (*asyncio.Queue* 的方法), 1116
shutdown() (*concurrent.futures.Executor* 的方法), 1032
shutdown() (*imaplib.IMAP4* 的方法), 1488
shutdown() (*multiprocessing.managers.BaseManager* 的方法), 1005
shutdown() (*queue.Queue* 的方法), 1067
shutdown() (*socketserver.BaseServer* 的方法), 1505
shutdown() (*socket.socket* 的方法), 1198
shutdown() (於 *logging* 模組中), 776

shutdown_asyncgens() (*asyncio.loop* 的方法), 1123
shutdown_default_executor() (*asyncio.loop* 的方法), 1123
shutil
    module, 492
SI (於 *curses.ascii* 模組中), 955
side_effect (*unittest.mock.Mock* 的屬性), 1783
SIG_BLOCK (於 *signal* 模組中), 1250
SIG_DFL (於 *signal* 模組中), 1248
SIG_IGN (於 *signal* 模組中), 1248
SIG_SETMASK (於 *signal* 模組中), 1250
SIG_UNBLOCK (於 *signal* 模組中), 1250
SIGABRT (於 *signal* 模組中), 1248
SIGALRM (於 *signal* 模組中), 1248
SIGBREAK (於 *signal* 模組中), 1248
SIGBUS (於 *signal* 模組中), 1248
SIGCHLD (於 *signal* 模組中), 1248
SIGCLD (於 *signal* 模組中), 1248
SIGCONT (於 *signal* 模組中), 1249
SIGFPE (於 *signal* 模組中), 1249
SIGHUP (於 *signal* 模組中), 1249
SIGILL (於 *signal* 模組中), 1249
SIGINT (於 *signal* 模組中), 1249
siginterrupt() (於 *signal* 模組中), 1253
SIGKILL (於 *signal* 模組中), 1249
Sigmasks (*signal* 中的類🄵), 1248
signal
    module, 1247
signal() (於 *signal* 模組中), 1253
Signals (*signal* 中的類🄵), 1248
signal (訊號)
    module (模組), 1073
Signature (*inspect* 中的類🄵), 2023
signature (*inspect.BoundArguments* 的屬性), 2026
signature() (於 *inspect* 模組中), 2022
sigpending() (於 *signal* 模組中), 1253
SIGPIPE (於 *signal* 模組中), 1249
SIGSEGV (於 *signal* 模組中), 1249
SIGSTKFLT (於 *signal* 模組中), 1249
SIGTERM (於 *signal* 模組中), 1249
sigtimedwait() (於 *signal* 模組中), 1254
SIGUSR1 (於 *signal* 模組中), 1249
SIGUSR2 (於 *signal* 模組中), 1249
sigwait() (於 *signal* 模組中), 1253
sigwaitinfo() (於 *signal* 模組中), 1253
SIGWINCH (於 *signal* 模組中), 1250
SIMPLE (*inspect.BufferFlags* 的屬性), 2034
Simple Mail Transfer Protocol (簡單郵件傳輸協定), 1490
SimpleCookie (*http.cookies* 中的類🄵), 1519
simplefilter() (於 *warnings* 模組中), 1968
SimpleHandler (*wsgiref.handlers* 中的類🄵), 1431
SimpleHTTPRequestHandler (*http.server* 中的類🄵), 1515
SimpleNamespace (*types* 中的類🄵), 308
SimpleQueue (*multiprocessing* 中的類🄵), 995
SimpleQueue (*queue* 中的類🄵), 1065
SimpleXMLRPCRequestHandler (*xmlrpc.server* 中的類🄵), 1538
SimpleXMLRPCServer (*xmlrpc.server* 中的類🄵), 1538
sin() (於 *cmath* 模組中), 351
sin() (於 *math* 模組中), 347
single dispatch (單一調度), **2241**
SingleAddressHeader (*email.headerregistry* 中的類🄵), 1288
singledispatch() (於 *functools* 模組中), 431
singledispatchmethod (*functools* 中的類🄵), 433
sinh() (於 *cmath* 模組中), 351
sinh() (於 *math* 模組中), 347
SIO_KEEPALIVE_VALS (於 *socket* 模組中), 1181
SIO_LOOPBACK_FAST_PATH (於 *socket* 模組中), 1181
SIO_RCVALL (於 *socket* 模組中), 1181
site
    module, 2043
site 命令列選項
    --user-base, 2046
    --user-site, 2046
site_maps() (*urllib.robotparser.RobotFileParser* 的方法), 1462
sitecustomize
    module, 2044
site-packages
    directory (目🄵), 2043
sixtofour (*ipaddress.IPv6Address* 的屬性), 1548
size (*ctypes.CField* 的屬性), 852
size (*multiprocessing.shared_memory.SharedMemory* 的屬性), 1027
size (*struct.Struct* 的屬性), 193
size (*tarfile.TarInfo* 的屬性), 610
size (*tracemalloc.Statistic* 的屬性), 1901
size (*tracemalloc.StatisticDiff* 的屬性), 1901
size (*tracemalloc.Trace* 的屬性), 1902
size() (*ftplib.FTP* 的方法), 1478
size() (*mmap.mmap* 的方法), 1259
size_diff (*tracemalloc.StatisticDiff* 的屬性), 1901
Sized (*collections.abc* 中的類🄵), 283
Sized (*typing* 中的類🄵), 1719
sizeof() (於 *ctypes* 模組中), 845
sizeof_digit (*sys.int_info* 的屬性), 1934
SKIP (於 *doctest* 模組中), 1732
skip() (於 *unittest* 模組中), 1754
skip_if_broken_multiprocessing_synchronize() (於 *test.support* 模組中), 1845
skip_unless_bind_unix_socket() (於 *test.support.socket_helper* 模組中), 1846
skip_unless_symlink() (於 *test.support.os_helper* 模組中), 1850
skip_unless_xattr() (於 *test.support.os_helper* 模組中), 1850
skipIf() (於 *unittest* 模組中), 1754
skipinitialspace (*csv.Dialect* 的屬性), 624
skipped (*doctest.TestResults* 的屬性), 1741
skipped (*unittest.TestResult* 的屬性), 1770

skippedEntity() (*xml.sax.handler.ContentHandler* 的方法), 1405
skips (*doctest.DocTestRunner* 的屬性), 1742
SkipTest, 1754
skipTest() (*unittest.TestCase* 的方法), 1756
skipUnless() (於 *unittest* 模組中), 1754
SLASH (於 *token* 模組中), 2136
SLASHEQUAL (於 *token* 模組中), 2137
sleep() (於 *asyncio* 模組中), 1085
sleep() (於 *time* 模組中), 751
sleeping_retry() (於 *test.support* 模組中), 1839
Slice (*ast* 中的類F), 2104
slice (F建類F), 27
slice（切片）, **2241**
    assignment（賦值）, 48
    built-in function（F建函式）, 2169
    operation（操作）, 46
slow_callback_duration (*asyncio.loop* 的屬性), 1137
SMALLEST (於 *test.support* 模組中), 1838
SMTP
    protocol（協定）, 1490
SMTP (*smtplib* 中的類F), 1490
SMTP (於 *email.policy* 模組中), 1283
SMTP_SSL (*smtplib* 中的類F), 1491
SMTPAuthenticationError, 1492
SMTPConnectError, 1492
smtpd
    module, 2222
SMTPDataError, 1492
SMTPException, 1491
SMTPHandler (*logging.handlers* 中的類F), 800
SMTPHeloError, 1492
smtplib
    module, 1490
SMTPNotSupportedError, 1492
SMTPRecipientsRefused, 1492
SMTPResponseException, 1492
SMTPSenderRefused, 1492
SMTPServerDisconnected, 1492
SMTPUTF8 (於 *email.policy* 模組中), 1283
Snapshot (*tracemalloc* 中的類F), 1900
SND_ALIAS (於 *winsound* 模組中), 2188
SND_APPLICATION (於 *winsound* 模組中), 2190
SND_ASYNC (於 *winsound* 模組中), 2189
SND_FILENAME (於 *winsound* 模組中), 2188
SND_LOOP (於 *winsound* 模組中), 2189
SND_MEMORY (於 *winsound* 模組中), 2189
SND_NODEFAULT (於 *winsound* 模組中), 2189
SND_NOSTOP (於 *winsound* 模組中), 2189
SND_NOWAIT (於 *winsound* 模組中), 2189
SND_PURGE (於 *winsound* 模組中), 2189
SND_SENTRY (於 *winsound* 模組中), 2190
SND_SYNC (於 *winsound* 模組中), 2190
SND_SYSTEM (於 *winsound* 模組中), 2190
sndhdr
    module, 2222
sni_callback (*ssl.SSLContext* 的屬性), 1222
sniff() (*csv.Sniffer* 的方法), 622
Sniffer (*csv* 中的類F), 622
SO (於 *curses.ascii* 模組中), 955
SO_HCI_EVT_FILTER (於 *socket* 模組中), 1182
SO_HCI_PKT_FILTER (於 *socket* 模組中), 1182
SO_INCOMING_CPU (於 *socket* 模組中), 1183
SO_REUSEPORT_LB (於 *socket* 模組中), 1183
sock_accept() (*asyncio.loop* 的方法), 1133
SOCK_CLOEXEC (於 *socket* 模組中), 1178
sock_connect() (*asyncio.loop* 的方法), 1132
SOCK_DGRAM (於 *socket* 模組中), 1178
SOCK_MAX_SIZE (於 *test.support* 模組中), 1838
SOCK_NONBLOCK (於 *socket* 模組中), 1178
SOCK_RAW (於 *socket* 模組中), 1178
SOCK_RDM (於 *socket* 模組中), 1178
sock_recv() (*asyncio.loop* 的方法), 1132
sock_recv_into() (*asyncio.loop* 的方法), 1132
sock_recvfrom() (*asyncio.loop* 的方法), 1132
sock_recvfrom_into() (*asyncio.loop* 的方法), 1132
sock_sendall() (*asyncio.loop* 的方法), 1132
sock_sendfile() (*asyncio.loop* 的方法), 1133
sock_sendto() (*asyncio.loop* 的方法), 1132
SOCK_SEQPACKET (於 *socket* 模組中), 1178
SOCK_STREAM (於 *socket* 模組中), 1178
socket
    module, 1173
    module（模組）, 1423
    object（物件）, 1174
socket (*socket* 中的類F), 1184
socket (*socketserver.BaseServer* 的屬性), 1506
socket() (*imaplib.IMAP4* 的方法), 1488
socket() (於 *socket* 模組), 1238
socket_type (*socketserver.BaseServer* 的屬性), 1506
SocketHandler (*logging.handlers* 中的類F), 795
socketpair() (於 *socket* 模組中), 1184
sockets (*asyncio.Server* 的屬性), 1140
socketserver
    module, 1503
SocketType (於 *socket* 模組中), 1186
soft deprecated（軟性F用）, **2241**
SOFT_KEYWORD (於 *token* 模組中), 2135
softkwlist (於 *keyword* 模組中), 2139
SOH (於 *curses.ascii* 模組中), 955
SOL_ALG (於 *socket* 模組中), 1181
SOL_BLUETOOTH (於 *socket* 模組中), 1182
SOL_HCI (於 *socket* 模組中), 1182
SOL_L2CAP (於 *socket* 模組中), 1182
SOL_RDS (於 *socket* 模組中), 1181
SOL_RFCOMM (於 *socket* 模組中), 1182
SOL_SCO (於 *socket* 模組中), 1182
SOMAXCONN (於 *socket* 模組中), 1178
sort() (*imaplib.IMAP4* 的方法), 1488
sort() (*list* 的方法), 49
sort_stats() (*pstats.Stats* 的方法), 1882
sortdict() (於 *test.support* 模組中), 1839
sorted()
    built-in function, 28
--sort-keys

    json 命令列選項, 1328
sortTestMethodsUsing (*unittest.TestLoader* 的屬性), 1769
source (*doctest.Example* 的屬性), 1739
source (*pdb command*), 1875
source (*shlex.shlex* 的屬性), 2194
SOURCE_DATE_EPOCH, 2146, 2148
source_from_cache() (於 *importlib.util* 模組中), 2074
source_hash() (於 *importlib.util* 模組中), 2075
SOURCE_SUFFIXES (於 *importlib.machinery* 模組中), 2068
source_to_code() (*importlib.abc.InspectLoader* 的🄵態方法), 2064
SourceFileLoader (*importlib.machinery* 中的類🄵), 2070
sourcehook() (*shlex.shlex* 的方法), 2193
SourcelessFileLoader (*importlib.machinery* 中的類🄵), 2070
SourceLoader (*importlib.abc* 中的類🄵), 2065
SP (於 *curses.ascii* 模組中), 956
space（空白）
    於 printf 風格格式化, 65, 81
--spacing
    calendar 命令列選項, 263
span() (*re.Match* 的方法), 155
sparse (*tarfile.TarInfo* 的屬性), 611
spawn() (於 *pty* 模組中), 2202
spawn_python() (於 *test.support.script_helper* 模組中), 1847
spawnl() (於 *os* 模組中), 722
spawnle() (於 *os* 模組中), 722
spawnlp() (於 *os* 模組中), 722
spawnlpe() (於 *os* 模組中), 722
spawnv() (於 *os* 模組中), 722
spawnve() (於 *os* 模組中), 722
spawnvp() (於 *os* 模組中), 722
spawnvpe() (於 *os* 模組中), 722
spec_from_file_location() (於 *importlib.util* 模組中), 2075
spec_from_loader() (於 *importlib.util* 模組中), 2075
special
    method（方法）, 2241
special method（特殊方法）, **2241**
SpecialFileError, 493, 605
--specialized
    dis 命令列選項, 2152
specified_attributes (*xml.parsers.expat.xmlparser* 的屬性), 1414
speed() (於 *turtle* 模組中), 1640
Spinbox (*tkinter.ttk* 中的類🄵), 1603
splice() (於 *os* 模組中), 687
SPLICE_F_MORE (於 *os* 模組中), 688
SPLICE_F_MOVE (於 *os* 模組中), 688
SPLICE_F_NONBLOCK (於 *os* 模組中), 688
split() (*BaseExceptionGroup* 的方法), 119
split() (*bytearray* 的方法), 74
split() (*bytes* 的方法), 74
split() (*re.Pattern* 的方法), 152
split() (*str* 的方法), 60
split() (於 *os.path* 模組中), 473
split() (於 *re* 模組中), 149
split() (於 *shlex* 模組中), 2191
splitdrive() (於 *os.path* 模組中), 474
splitext() (於 *os.path* 模組中), 474
splitlines() (*bytearray* 的方法), 78
splitlines() (*bytes* 的方法), 78
splitlines() (*str* 的方法), 61
SplitResult (*urllib.parse* 中的類🄵), 1458
SplitResultBytes (*urllib.parse* 中的類🄵), 1459
splitroot() (於 *os.path* 模組中), 474
SpooledTemporaryFile (*tempfile* 中的類🄵), 484
sprintf 風格格式化, 64, 80
spwd
    module, 2222
sqlite3
    module, 531
SQLITE_DBCONFIG_DEFENSIVE (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_DQS_DDL (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_DQS_DML (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_FKEY (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_QPSG (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_TRIGGER (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_ENABLE_VIEW (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_LEGACY_ALTER_TABLE (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_LEGACY_FILE_FORMAT (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_RESET_DATABASE (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_TRIGGER_EQP (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_TRUSTED_SCHEMA (於 *sqlite3* 模組中), 537
SQLITE_DBCONFIG_WRITABLE_SCHEMA (於 *sqlite3* 模組中), 537
SQLITE_DENY (於 *sqlite3* 模組中), 536
sqlite_errorcode (*sqlite3.Error* 的屬性), 552
sqlite_errorname (*sqlite3.Error* 的屬性), 552
SQLITE_IGNORE (於 *sqlite3* 模組中), 536
SQLITE_OK (於 *sqlite3* 模組中), 536
sqlite_version (於 *sqlite3* 模組中), 536
sqlite_version_info (於 *sqlite3* 模組中), 536
sqrt() (*decimal.Context* 的方法), 370

sqrt() (*decimal.Decimal* 的方法), 363
sqrt() (於 *cmath* 模組中), 351
sqrt() (於 *math* 模組中), 345
SSL, 1202
ssl
    module, 1202
ssl_version (*ftplib.FTP_TLS* 的屬性), 1479
SSLCertVerificationError, 1205
SSLContext (*ssl* 中的類🄵), 1218
SSLEOFError, 1205
SSLError, 1205
SSLErrorNumber (*ssl* 中的類🄵), 1213
SSLKEYLOGFILE, 1204, 1205
SSLObject (*ssl* 中的類🄵), 1233
sslobject_class (*ssl.SSLContext* 的屬性), 1224
SSLSession (*ssl* 中的類🄵), 1234
SSLSocket (*ssl* 中的類🄵), 1214
sslsocket_class (*ssl.SSLContext* 的屬性), 1223
SSLSyscallError, 1205
SSLv3 (*ssl.TLSVersion* 的屬性), 1214
SSLWantReadError, 1205
SSLWantWriteError, 1205
SSLZeroReturnError, 1205
st() (於 *turtle* 模組中), 1646
st_atime (*os.stat_result* 的屬性), 703
ST_ATIME (於 *stat* 模組中), 477
st_atime_ns (*os.stat_result* 的屬性), 703
st_birthtime (*os.stat_result* 的屬性), 703
st_birthtime_ns (*os.stat_result* 的屬性), 704
st_blksize (*os.stat_result* 的屬性), 704
st_blocks (*os.stat_result* 的屬性), 704
st_creator (*os.stat_result* 的屬性), 704
st_ctime (*os.stat_result* 的屬性), 703
ST_CTIME (於 *stat* 模組中), 477
st_ctime_ns (*os.stat_result* 的屬性), 703
st_dev (*os.stat_result* 的屬性), 703
ST_DEV (於 *stat* 模組中), 476
st_file_attributes (*os.stat_result* 的屬性), 705
st_flags (*os.stat_result* 的屬性), 704
st_fstype (*os.stat_result* 的屬性), 704
st_gen (*os.stat_result* 的屬性), 704
st_gid (*os.stat_result* 的屬性), 703
ST_GID (於 *stat* 模組中), 477
st_ino (*os.stat_result* 的屬性), 703
ST_INO (於 *stat* 模組中), 476
st_mode (*os.stat_result* 的屬性), 703
ST_MODE (於 *stat* 模組中), 476
st_mtime (*os.stat_result* 的屬性), 703
ST_MTIME (於 *stat* 模組中), 477
st_mtime_ns (*os.stat_result* 的屬性), 703
st_nlink (*os.stat_result* 的屬性), 703
ST_NLINK (於 *stat* 模組中), 476
st_rdev (*os.stat_result* 的屬性), 704
st_reparse_tag (*os.stat_result* 的屬性), 705
st_rsize (*os.stat_result* 的屬性), 704
st_size (*os.stat_result* 的屬性), 703
ST_SIZE (於 *stat* 模組中), 477
st_type (*os.stat_result* 的屬性), 704
st_uid (*os.stat_result* 的屬性), 703
ST_UID (於 *stat* 模組中), 476
stack (*traceback.TracebackException* 的屬性), 2003
stack viewer（堆🄵檢視器）, 1620
stack() (於 *inspect* 模組中), 2031
stack_effect() (於 *dis* 模組中), 2155
stack_size() (於 *_thread* 模組中), 1072
stack_size() (於 *threading* 模組中), 968
stackable（可堆🄵）
    streams（串流）, 194
StackSummary (*traceback* 中的類🄵), 2005
stamp() (於 *turtle* 模組中), 1639
standard_b64decode() (於 *base64* 模組中), 1353
standard_b64encode() (於 *base64* 模組中), 1353
standend() (*curses.window* 的方法), 941
standout() (*curses.window* 的方法), 941
STAR (於 *token* 模組中), 2136
STAREQUAL (於 *token* 模組中), 2137
starmap() (*multiprocessing.pool.Pool* 的方法), 1012
starmap() (於 *itertools* 模組中), 418
starmap_async() (*multiprocessing.pool.Pool* 的方法), 1012
Starred (*ast* 中的類🄵), 2101
start (*range* 的屬性), 51
start (*slice* 的屬性), 28
start (*UnicodeError* 的屬性), 115
start() (*logging.handlers.QueueListener* 的方法), 804
start() (*multiprocessing.managers.BaseManager* 的方法), 1005
start() (*multiprocessing.Process* 的方法), 990
start() (*re.Match* 的方法), 155
start() (*threading.Thread* 的方法), 972
start() (*tkinter.ttk.Progressbar* 的方法), 1606
start() (於 *tracemalloc* 模組中), 1898
start() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
start_color() (於 *curses* 模組中), 934
start_new_thread() (於 *_thread* 模組中), 1072
start_ns() (*xml.etree.ElementTree.TreeBuilder* 的方法), 1381
start_server() (於 *asyncio* 模組中), 1099
start_serving() (*asyncio.Server* 的方法), 1140
start_threads() (於 *test.support.threading_helper* 模組中), 1848
start_tls() (*asyncio.loop* 的方法), 1131
start_tls() (*asyncio.StreamWriter* 的方法), 1102
start_trace() (*bdb.Bdb* 的方法), 1861
start_unix_server() (於 *asyncio* 模組中), 1100
StartBoundaryNotFoundDefect, 1285
startCDATA() (*xml.sax.handler.LexicalHandler* 的方法), 1406
StartCdataSectionHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
--start-directory
    unittest-discover 命令列選項, 1750
StartDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* 的方法),

1415
startDocument() (*xml.sax.handler.ContentHandler* 的方法), 1403
startDTD() (*xml.sax.handler.LexicalHandler* 的方法), 1406
startElement() (*xml.sax.handler.ContentHandler* 的方法), 1404
StartElementHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
startElementNS() (*xml.sax.handler.ContentHandler* 的方法), 1404
STARTF_FORCEOFFFEEDBACK (於 *subprocess* 模組中), 1056
STARTF_FORCEONFEEDBACK (於 *subprocess* 模組中), 1056
STARTF_USESHOWWINDOW (於 *subprocess* 模組中), 1056
STARTF_USESTDHANDLES (於 *subprocess* 模組中), 1056
startfile() (於 *os* 模組中), 724
StartNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1416
startPrefixMapping() (*xml.sax.handler.ContentHandler* 的方法), 1404
StartResponse (*wsgiref.types* 中的類🄵), 1433
startswith() (*bytearray* 的方法), 72
startswith() (*bytes* 的方法), 72
startswith() (*str* 的方法), 61
startTest() (*unittest.TestResult* 的方法), 1771
startTestRun() (*unittest.TestResult* 的方法), 1771
starttls() (*imaplib.IMAP4* 的方法), 1488
starttls() (*smtplib.SMTP* 的方法), 1494
STARTUPINFO (*subprocess* 中的類🄵), 1055
stat
    module, 475
    module（模組）, 702
stat() (*os.DirEntry* 的方法), 701
stat() (*pathlib.Path* 的方法), 457
stat() (*poplib.POP3* 的方法), 1481
stat() (於 *os* 模組中), 702
stat_result (*os* 中的類🄵), 702
state() (*tkinter.ttk.Widget* 的方法), 1602
statement（陳述式）, **2241**
    assert, 111
    del, 48, 91
    except, 109
    if, 37
    import（引入）, 32, 2043
    raise, 109
    try, 109
    while, 37
static type checker（🄵態型🄵檢查器）, **2241**
static_order() (*graphlib.TopologicalSorter* 的方法), 336
staticmethod()
    built-in function, 28
Statistic (*tracemalloc* 中的類🄵), 1901
StatisticDiff (*tracemalloc* 中的類🄵), 1901
statistics
    module, 393
statistics() (*tracemalloc.Snapshot* 的方法), 1900
StatisticsError, 403
Stats (*pstats* 中的類🄵), 1881
status (*http.client.HTTPResponse* 的屬性), 1472
status (*urllib.response.addinfourl* 的屬性), 1452
status() (*imaplib.IMAP4* 的方法), 1489
statvfs() (於 *os* 模組中), 705
STD_ERROR_HANDLE (於 *subprocess* 模組中), 1056
STD_INPUT_HANDLE (於 *subprocess* 模組中), 1056
STD_OUTPUT_HANDLE (於 *subprocess* 模組中), 1056
stderr (*asyncio.subprocess.Process* 的屬性), 1114
stderr (*subprocess.CalledProcessError* 的屬性), 1047
stderr (*subprocess.CompletedProcess* 的屬性), 1045
stderr (*subprocess.Popen* 的屬性), 1054
stderr (*subprocess.TimeoutExpired* 的屬性), 1046
stderr (於 *sys* 模組中), 1943
stdev (*statistics.NormalDist* 的屬性), 404
stdev() (於 *statistics* 模組中), 400
stdin (*asyncio.subprocess.Process* 的屬性), 1114
stdin (*subprocess.Popen* 的屬性), 1054
stdin (於 *sys* 模組中), 1943
stdlib_module_names (於 *sys* 模組中), 1944
stdlib（標准函式庫）, **2241**
stdout (*asyncio.subprocess.Process* 的屬性), 1114
stdout (*subprocess.CalledProcessError* 的屬性), 1046
stdout (*subprocess.CompletedProcess* 的屬性), 1045
stdout (*subprocess.Popen* 的屬性), 1054
stdout (*subprocess.TimeoutExpired* 的屬性), 1046
STDOUT (於 *subprocess* 模組中), 1046
stdout (於 *sys* 模組中), 1943
stem (*pathlib.PurePath* 的屬性), 450
step (*pdb command*), 1874
step (*range* 的屬性), 51
step (*slice* 的屬性), 28
step() (*tkinter.ttk.Progressbar* 的方法), 1606
stls() (*poplib.POP3* 的方法), 1482
stop (*range* 的屬性), 51
stop (*slice* 的屬性), 28
stop() (*asyncio.loop* 的方法), 1122
stop() (*logging.handlers.QueueListener* 的方法), 804
stop() (*tkinter.ttk.Progressbar* 的方法), 1606
stop() (*unittest.TestResult* 的方法), 1771
stop() (於 *tracemalloc* 模組中), 1898
stop_here() (*bdb.Bdb* 的方法), 1862
STOP_ITERATION (*monitoring event*), 1948
stop_trace() (*bdb.Bdb* 的方法), 1861
StopAsyncIteration, 114
StopIteration, 113
stopListening() (於 *logging.config* 模組中), 780
stopTest() (*unittest.TestResult* 的方法), 1771
stopTestRun() (*unittest.TestResult* 的方法), 1771
storbinary() (*ftplib.FTP* 的方法), 1476
Store (*ast* 中的類🄵), 2101

store() (*imaplib.IMAP4* 的方法), 1489
STORE_ACTIONS (*optparse.Option* 的屬性), 923
STORE_ATTR (*opcode*), 2162
STORE_DEREF (*opcode*), 2168
STORE_FAST (*opcode*), 2167
STORE_FAST_LOAD_FAST (*opcode*), 2167
STORE_FAST_STORE_FAST (*opcode*), 2167
STORE_GLOBAL (*opcode*), 2163
STORE_NAME (*opcode*), 2162
STORE_SLICE (*opcode*), 2159
STORE_SUBSCR (*opcode*), 2159
storlines() (*ftplib.FTP* 的方法), 1476
str (Ⓕ建類Ⓕ), 53
str() (於 *locale* 模組中), 1574
str_digits_check_threshold (*sys.int_info* 的屬性), 1934
Strategy (*compression.zstd* 中的類Ⓕ), 572
strategy (*compression.zstd.CompressionParameter* 的屬性), 570
strcoll() (於 *locale* 模組中), 1574
StreamError, 604
StreamHandler (*logging* 中的類Ⓕ), 790
StreamReader (*asyncio* 中的類Ⓕ), 1100
StreamReader (*codecs* 中的類Ⓕ), 202
streamreader (*codecs.CodecInfo* 的屬性), 194
StreamReaderWriter (*codecs* 中的類Ⓕ), 203
StreamRecoder (*codecs* 中的類Ⓕ), 203
StreamRequestHandler (*socketserver* 中的類Ⓕ), 1507
streams（串流）, 194
    stackable（可堆Ⓕ）, 194
StreamWriter (*asyncio* 中的類Ⓕ), 1101
StreamWriter (*codecs* 中的類Ⓕ), 201
streamwriter (*codecs.CodecInfo* 的屬性), 194
StrEnum (*enum* 中的類Ⓕ), 326
strerror (*OSError* 的屬性), 113
strerror() (於 *os* 模組中), 675
strftime() (*datetime.date* 的方法), 223
strftime() (*datetime.datetime* 的方法), 233
strftime() (*datetime.time* 的方法), 238
strftime() (於 *time* 模組中), 752
strict
    error handler's name（錯誤處理器名稱）, 197
strict (*csv.Dialect* 的屬性), 624
STRICT (*enum.FlagBoundary* 的屬性), 330
strict (於 *email.policy* 模組中), 1283
strict_domain (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1527
strict_errors() (於 *codecs* 模組中), 198
strict_ns_domain (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
strict_ns_set_initial_dollar (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
strict_ns_set_path (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
strict_ns_unverifiable (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1528
strict_rfc2965_unverifiable (*http.cookiejar.DefaultCookiePolicy* 的屬性), 1527
STRIDED (*inspect.BufferFlags* 的屬性), 2034
STRIDED_RO (*inspect.BufferFlags* 的屬性), 2034
STRIDES (*inspect.BufferFlags* 的屬性), 2034
strides (*memoryview* 的屬性), 89
string
    module, 123
STRING (*annotationlib.Format* 的屬性), 2037
string (*re.Match* 的屬性), 155
STRING (於 *token* 模組中), 2134
string_at() (於 *ctypes* 模組中), 845
StringIO (*io* 中的類Ⓕ), 746
stringprep
    module, 179
strings (*string.templatelib.Template* 的屬性), 135
string.templatelib
    module, 134
string（字串）
    format()（Ⓕ建函式）, 16
    formatted literal（格式化常數）, 63
    formatting（格式化）、printf, 64
    interpolated literal（插值常數）, 63
    interpolation（插值）、printf, 64
    methods（方法）, 54
    object（物件）, 52
    str()（Ⓕ建函式）, 29
    str（Ⓕ建類Ⓕ）, 53
    text sequence type（文字序列型Ⓕ）, 52
strip() (*bytearray* 的方法), 75
strip() (*bytes* 的方法), 75
strip() (*str* 的方法), 61
strip_dirs() (*pstats.Stats* 的方法), 1882
stripspaces (*curses.textpad.Textbox* 的屬性), 954
strong reference（Ⓕ參照）, **2241**
strptime() (*datetime.date* 的類Ⓕ方法), 220
strptime() (*datetime.datetime* 的類Ⓕ方法), 227
strptime() (*datetime.time* 的類Ⓕ方法), 237
strptime() (於 *time* 模組中), 754
strsignal() (於 *signal* 模組中), 1251
struct
    module, 187
    module（模組）, 1198
Struct (*struct* 中的類Ⓕ), 193
struct_time (*time* 中的類Ⓕ), 754
Structure (*ctypes* 中的類Ⓕ), 849
structures（結構）
    C, 187
strxfrm() (於 *locale* 模組中), 1574
str（Ⓕ建類Ⓕ）
    （亦請見 string）, 52
STX (於 *curses.ascii* 模組中), 955
Style (*tkinter.ttk* 中的類Ⓕ), 1612
Sub (*ast* 中的類Ⓕ), 2102

SUB (於 *curses.ascii* 模組中), 956
sub() (*re.Pattern* 的方法), 153
sub() (於 *operator* 模組中), 437
sub() (於 *re* 模組中), 150
subdirs (*filecmp.dircmp* 的屬性), 482
SubElement() (於 *xml.etree.ElementTree* 模組中), 1374
subgroup() (*BaseExceptionGroup* 的方法), 119
submit() (*concurrent.futures.Executor* 的方法), 1032
submodule_search_locations (*importlib.machinery.ModuleSpec* 的屬性), 2072
subn() (*re.Pattern* 的方法), 153
subn() (於 *re* 模組中), 151
subnet_of() (*ipaddress.IPv4Network* 的方法), 1551
subnet_of() (*ipaddress.IPv6Network* 的方法), 1553
subnets() (*ipaddress.IPv4Network* 的方法), 1551
subnets() (*ipaddress.IPv6Network* 的方法), 1553
Subnormal (*decimal* 中的類🄵), 373
suboffsets (*memoryview* 的屬性), 89
subpad() (*curses.window* 的方法), 941
subprocess
    module, 1044
subprocess_exec() (*asyncio.loop* 的方法), 1137
subprocess_shell() (*asyncio.loop* 的方法), 1138
SubprocessError, 1046
SubprocessProtocol (*asyncio* 中的類🄵), 1152
SubprocessTransport (*asyncio* 中的類🄵), 1148
subscribe() (*imaplib.IMAP4* 的方法), 1489
Subscript (*ast* 中的類🄵), 2104
subscript (下標)
    assignment (賦值), 48
    operation (操作), 46
subsequent_indent (*textwrap.TextWrapper* 的屬性), 175
substitute() (*string.Template* 的方法), 132
subTest() (*unittest.TestCase* 的方法), 1756
subtract() (*collections.Counter* 的方法), 267
subtract() (*decimal.Context* 的方法), 370
subtype (*email.headerregistry.ContentTypeHeader* 的屬性), 1289
subwin() (*curses.window* 的方法), 941
successful() (*multiprocessing.pool.AsyncResult* 的方法), 1012
suffix (*pathlib.PurePath* 的屬性), 449
suffix_map (*mimetypes.MimeTypes* 的屬性), 1350
suffix_map (於 *mimetypes* 模組中), 1349
suffixes (*pathlib.PurePath* 的屬性), 449
suiteClass (*unittest.TestLoader* 的屬性), 1769
sum()
    built-in function, 29
summarize() (*doctest.DocTestRunner* 的方法), 1742
summarize_address_range() (於 *ipaddress* 模組中), 1556
--summary
    trace 命令列選項, 1891
sumprod() (於 *math* 模組中), 346
sunau
    module, 2222
SUNDAY (於 *calendar* 模組中), 260
super (*pyclbr.Class* 的屬性), 2145
super (🄵建類🄵), 29
supernet() (*ipaddress.IPv4Network* 的方法), 1551
supernet() (*ipaddress.IPv6Network* 的方法), 1553
supernet_of() (*ipaddress.IPv4Network* 的方法), 1552
supernet_of() (*ipaddress.IPv6Network* 的方法), 1553
supports_bytes_environ (於 *os* 模組中), 675
supports_dir_fd (於 *os* 模組中), 706
supports_effective_ids (於 *os* 模組中), 706
supports_fd (於 *os* 模組中), 706
supports_follow_symlinks (於 *os* 模組中), 706
supports_unicode_filenames (於 *os.path* 模組中), 475
SupportsAbs (*typing* 中的類🄵), 1705
SupportsBytes (*typing* 中的類🄵), 1705
SupportsComplex (*typing* 中的類🄵), 1705
SupportsFloat (*typing* 中的類🄵), 1705
SupportsIndex (*typing* 中的類🄵), 1705
SupportsInt (*typing* 中的類🄵), 1705
SupportsRound (*typing* 中的類🄵), 1705
suppress() (於 *contextlib* 模組中), 1984
SuppressCrashReport (*test.support* 中的類🄵), 1845
surrogateescape
    error handler's name (錯誤處理器名稱), 197
surrogatepass
    error handler's name (錯誤處理器名稱), 198
SW_HIDE (於 *subprocess* 模組中), 1056
SWAP (*opcode*), 2158
swap_attr() (於 *test.support* 模組中), 1841
swap_item() (於 *test.support* 模組中), 1841
swapcase() (*bytearray* 的方法), 78
swapcase() (*bytes* 的方法), 78
swapcase() (*str* 的方法), 62
Symbol (*symtable* 中的類🄵), 2132
SymbolTable (*symtable* 中的類🄵), 2130
SymbolTableType (*symtable* 中的類🄵), 2130
symlink() (於 *os* 模組中), 707
symlink_to() (*pathlib.Path* 的方法), 463
--symlinks
    venv 命令列選項, 1906
symmetric_difference() (*frozenset* 的方法), 90
symmetric_difference_update() (*frozenset* 的方法), 91
symtable
    module, 2130
symtable() (於 *symtable* 模組中), 2130
SYMTYPE (於 *tarfile* 模組中), 605
SYN (於 *curses.ascii* 模組中), 956
sync() (*dbm.dumb.dumbdbm* 的方法), 531
sync() (*dbm.gnu.gdbm* 的方法), 529
sync() (*shelve.Shelf* 的方法), 522
sync() (於 *os* 模組中), 707

syncdown() (*curses.window* 的方法), 941
synchronized() (於 *multiprocessing.sharedctypes* 模組中), 1003
SyncManager (*multiprocessing.managers* 中的類🄵), 1006
syncok() (*curses.window* 的方法), 941
syncup() (*curses.window* 的方法), 941
SyntaxErr, 1392
SyntaxError, 114
SyntaxWarning, 118
sys
    module, 1919
    module（模組）, 24
sys_version (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
sysconf() (於 *os* 模組中), 731
sysconf_names (於 *os* 模組中), 731
sysconfig
    module, 1951
syslog
    module, 2210
syslog() (於 *syslog* 模組中), 2210
SysLogHandler (*logging.handlers* 中的類🄵), 797
sys.monitoring
    module, 1946
system() (於 *os* 模組中), 724
system() (於 *platform* 模組中), 805
system_alias() (於 *platform* 模組中), 805
system_must_validate_cert() (於 *test.support* 模組中), 1842
SystemError, 114
SystemExit, 115
systemId (*xml.dom.DocumentType* 的屬性), 1388
SystemRandom (*random* 中的類🄵), 389
SystemRandom (*secrets* 中的類🄵), 664
SystemRoot, 1051
--system-site-packages
    venv 命令列選項, 1906

T

-T
    trace 命令列選項, 1891
-t
    calendar 命令列選項, 263
    idle 命令列選項, 1624
    tarfile 命令列選項, 616
    trace 命令列選項, 1891
    unittest-discover 命令列選項, 1750
    webbrowser 命令列選項, 1423
    zipfile 命令列選項, 601
t-strings（t 字串）, **2242**
t-string（t 字串）, **2242**
T_FMT (於 *locale* 模組中), 1571
T_FMT_AMPM (於 *locale* 模組中), 1571
--tab
    json 命令列選項, 1329
TAB (於 *curses.ascii* 模組中), 955
tab() (*tkinter.ttk.Notebook* 的方法), 1605
TabError, 114
tabnanny
    module, 2143
tabs() (*tkinter.ttk.Notebook* 的方法), 1605
tabsize (*textwrap.TextWrapper* 的屬性), 175
tabular（表格）
    data（資料）, 619
tag (*xml.etree.ElementTree.Element* 的屬性), 1377
tag_bind() (*tkinter.ttk.Treeview* 的方法), 1612
tag_configure() (*tkinter.ttk.Treeview* 的方法), 1612
tag_has() (*tkinter.ttk.Treeview* 的方法), 1612
tagName (*xml.dom.Element* 的屬性), 1390
tail (*xml.etree.ElementTree.Element* 的屬性), 1377
take_snapshot() (於 *tracemalloc* 模組中), 1898
takewhile() (於 *itertools* 模組中), 418
tan() (於 *cmath* 模組中), 351
tan() (於 *math* 模組中), 347
tanh() (於 *cmath* 模組中), 351
tanh() (於 *math* 模組中), 347
tar_filter() (於 *tarfile* 模組中), 613
TarError, 604
tarfile
    module, 602
TarFile (*tarfile* 中的類🄵), 606
tarfile 命令列選項
    -c, 616
    --create, 616
    -e, 616
    --extract, 616
    --filter, 616
    -l, 616
    --list, 616
    -t, 616
    --test, 616
    -v, 616
    --verbose, 616
target (*xml.dom.ProcessingInstruction* 的屬性), 1391
target_length (*compression.zstd.CompressionParameter* 的屬性), 570
TarInfo (*tarfile* 中的類🄵), 610
tarinfo (*tarfile.FilterError* 的屬性), 604
Task (*asyncio* 中的類🄵), 1094
task_done() (*asyncio.Queue* 的方法), 1116
task_done() (*multiprocessing.JoinableQueue* 的方法), 995
task_done() (*queue.Queue* 的方法), 1066
TaskGroup (*asyncio* 中的類🄵), 1083
tau (於 *cmath* 模組中), 352
tau (於 *math* 模組中), 348
tb_locals (*unittest.TestResult* 的屬性), 1771
tbreak (*pdb command*), 1873
tcdrain() (於 *termios* 模組中), 2199
tcflow() (於 *termios* 模組中), 2199
tcflush() (於 *termios* 模組中), 2199
tcgetattr() (於 *termios* 模組中), 2199
tcgetpgrp() (於 *os* 模組中), 688
tcgetwinsize() (於 *termios* 模組中), 2200

Tcl() (於 tkinter 模組中), 1581
TCPServer (socketserver 中的類F), 1503
TCSADRAIN (於 termios 模組中), 2199
TCSAFLUSH (於 termios 模組中), 2199
TCSANOW (於 termios 模組中), 2199
tcsendbreak() (於 termios 模組中), 2199
tcsetattr() (於 termios 模組中), 2199
tcsetpgrp() (於 os 模組中), 688
tcsetwinsize() (於 termios 模組中), 2200
tearDown() (unittest.TestCase 的方法), 1756
tearDownClass() (unittest.TestCase 的方法), 1756
tearDownModule() (於 unittest 模組中), 1775
tee() (於 itertools 模組中), 418
teleport() (於 turtle 模組中), 1637
tell() (io.IOBase 的方法), 738
tell() (io.TextIOBase 的方法), 744
tell() (io.TextIOWrapper 的方法), 745
tell() (mmap.mmap 的方法), 1259
tell() (sqlite3.Blob 的方法), 551
tell() (wave.Wave_read 的方法), 1558
tell() (wave.Wave_write 的方法), 1559
telnetlib
    module, 2223
TEMP, 486
temp_cwd() (於 test.support.os_helper 模組中), 1850
temp_dir() (於 test.support.os_helper 模組中), 1850
temp_umask() (於 test.support.os_helper 模組中), 1850
tempdir (於 tempfile 模組中), 486
tempfile
    module, 483
Template (string 中的類F), 132
template (string.Template 的屬性), 133
Template (string.templatelib 中的類F), 135
TemplateStr (ast 中的類F), 2099
temporary (bdb.Breakpoint 的屬性), 1860
TemporaryDirectory (tempfile 中的類F), 484
TemporaryFile() (於 tempfile 模組中), 483
temporary (臨時)
    file name (檔案名稱), 483
    file (檔案), 483
teredo (ipaddress.IPv6Address 的屬性), 1548
TERM, 934
termattrs() (於 curses 模組中), 934
terminal_size (os 中的類F), 689
terminate() (asyncio.subprocess.Process 的方法), 1114
terminate() (asyncio.SubprocessTransport 的方法), 1151
terminate() (multiprocessing.pool.Pool 的方法), 1012
terminate() (multiprocessing.Process 的方法), 991
terminate() (subprocess.Popen 的方法), 1054
terminate_workers() (concurrent.futures.ProcessPoolExecutor 的方法), 1036
terminator (logging.StreamHandler 的屬性), 791
termios
    module, 2199
termname() (於 curses 模組中), 934
--terse
    platform 命令列選項, 808
test
    module, 1834
--test
    tarfile 命令列選項, 616
    zipfile 命令列選項, 601
test (doctest.DocTestFailure 的屬性), 1745
test (doctest.UnexpectedException 的屬性), 1745
TEST_DATA_DIR (於 test.support 模組中), 1838
TEST_HOME_DIR (於 test.support 模組中), 1838
TEST_HTTP_URL (於 test.support 模組中), 1838
TEST_SUPPORT_DIR (於 test.support 模組中), 1838
TestCase (unittest 中的類F), 1755
TestFailed, 1836
testfile() (於 doctest 模組中), 1734
TESTFN (於 test.support.os_helper 模組中), 1849
TESTFN_NONASCII (於 test.support.os_helper 模組中), 1849
TESTFN_UNDECODABLE (於 test.support.os_helper 模組中), 1849
TESTFN_UNENCODABLE (於 test.support.os_helper 模組中), 1849
TESTFN_UNICODE (於 test.support.os_helper 模組中), 1849
TestLoader (unittest 中的類F), 1767
testMethodPrefix (unittest.TestLoader 的屬性), 1769
testmod() (於 doctest 模組中), 1735
testNamePatterns (unittest.TestLoader 的屬性), 1770
test.regrtest
    module, 1836
TestResult (unittest 中的類F), 1770
TestResults (doctest 中的類F), 1741
testsource() (於 doctest 模組中), 1744
testsRun (unittest.TestResult 的屬性), 1770
TestSuite (unittest 中的類F), 1767
test.support
    module, 1836
test.support.bytecode_helper
    module, 1847
test.support.import_helper
    module, 1851
test.support.os_helper
    module, 1849
test.support.script_helper
    module, 1846
test.support.socket_helper
    module, 1846
test.support.threading_helper
    module, 1848
test.support.warnings_helper
    module, 1852
testzip() (zipfile.ZipFile 的方法), 595
text (ctypes.COMError 的屬性), 853

text (*SyntaxError* 的屬性), 114
text (*traceback.TracebackException* 的屬性), 2004
Text (*typing* 中的類F), 1716
text (*xml.etree.ElementTree.Element* 的屬性), 1377
text encoding（文字編碼）, **2242**
text file（文字檔案）, **2242**
text mode（文字模式）, 24
text_encoding() (於 *io* 模組中), 736
text_factory (*sqlite3.Connection* 的屬性), 547
Textbox (*curses.textpad* 中的類F), 953
TextCalendar (*calendar* 中的類F), 257
textdomain() (於 *gettext* 模組中), 1561
textdomain() (於 *locale* 模組中), 1577
textinput() (於 *turtle* 模組中), 1656
TextIO (*typing* 中的類F), 1705
TextIOBase (*io* 中的類F), 743
TextIOWrapper (*io* 中的類F), 744
TextTestResult (*unittest* 中的類F), 1772
TextTestRunner (*unittest* 中的類F), 1772
textwrap
    module, 173
TextWrapper (*textwrap* 中的類F), 174
TFD_CLOEXEC (於 *os* 模組中), 714
TFD_NONBLOCK (於 *os* 模組中), 714
TFD_TIMER_ABSTIME (於 *os* 模組中), 714
TFD_TIMER_CANCEL_ON_SET (於 *os* 模組中), 714
theme_create() (*tkinter.ttk.Style* 的方法), 1615
theme_names() (*tkinter.ttk.Style* 的方法), 1616
theme_settings() (*tkinter.ttk.Style* 的方法), 1615
theme_use() (*tkinter.ttk.Style* 的方法), 1616
THOUSEP (於 *locale* 模組中), 1572
Thread (*threading* 中的類F), 971
thread state, **2242**
thread() (*imaplib.IMAP4* 的方法), 1489
thread_info (於 *sys* 模組中), 1944
thread_inherit_context (*sys.flags* 的屬性), 1926
thread_time() (於 *time* 模組中), 756
thread_time_ns() (於 *time* 模組中), 756
thread_unsafe() (於 *test.support* 模組中), 1842
threading
    module, 965
threading_cleanup() (於 *test.support.threading_helper* 模組中), 1848
threading_setup() (於 *test.support.threading_helper* 模組中), 1848
ThreadingHTTPServer (*http.server* 中的類F), 1511
ThreadingHTTPSServer (*http.server* 中的類F), 1512
ThreadingMixIn (*socketserver* 中的類F), 1504
ThreadingMock (*unittest.mock* 中的類F), 1792
ThreadingTCPServer (*socketserver* 中的類F), 1504
ThreadingUDPServer (*socketserver* 中的類F), 1504
ThreadingUnixDatagramServer (*socketserver* 中的類F), 1504
ThreadingUnixStreamServer (*socketserver* 中的類F), 1504
ThreadPool (*multiprocessing.pool* 中的類F), 1017
ThreadPoolExecutor (*concurrent.futures* 中的類F), 1033
threadsafety (於 *sqlite3* 模組中), 537
threads（執行緒）
    POSIX, 1071
THURSDAY (於 *calendar* 模組中), 260
ticket_lifetime_hint (*ssl.SSLSession* 的屬性), 1234
tigetflag() (於 *curses* 模組中), 934
tigetnum() (於 *curses* 模組中), 934
tigetstr() (於 *curses* 模組中), 934
TILDE (於 *token* 模組中), 2137
tilt() (於 *turtle* 模組中), 1648
tiltangle() (於 *turtle* 模組中), 1648
time
    module, 748
time (*datetime* 中的類F), 236
time (*ssl.SSLSession* 的屬性), 1234
time (*uuid.UUID* 的屬性), 1498
time() (*asyncio.loop* 的方法), 1124
time() (*datetime.datetime* 的方法), 230
time() (於 *time* 模組中), 755
Time2Internaldate() (於 *imaplib* 模組中), 1484
time_hi_version (*uuid.UUID* 的屬性), 1498
time_low (*uuid.UUID* 的屬性), 1498
time_mid (*uuid.UUID* 的屬性), 1498
time_ns() (於 *time* 模組中), 756
timedelta (*datetime* 中的類F), 215
TimedRotatingFileHandler (*logging.handlers* 中的類F), 794
timegm() (於 *calendar* 模組中), 260
timeit
    module, 1886
timeit 命令列選項
    -h, 1888
    --help, 1888
    -n, 1888
    --number, 1888
    -p, 1888
    --process, 1888
    -r, 1888
    --repeat, 1888
    -s, 1888
    --setup, 1888
    -u, 1888
    --unit, 1888
    -v, 1888
    --verbose, 1888
timeit() (*timeit.Timer* 的方法), 1887
timeit() (於 *timeit* 模組中), 1886
timeout, 1177
Timeout (*asyncio* 中的類F), 1088
timeout (*socketserver.BaseServer* 的屬性), 1506
timeout (*ssl.SSLSession* 的屬性), 1234
timeout (*subprocess.TimeoutExpired* 的屬性), 1046
timeout() (*curses.window* 的方法), 941
timeout() (於 *asyncio* 模組中), 1088
timeout_at() (於 *asyncio* 模組中), 1089

TIMEOUT_MAX (於 _thread 模組中), 1073
TIMEOUT_MAX (於 threading 模組中), 968
TimeoutError, 117, 992, 1039, 1118
TimeoutExpired, 1046
Timer (threading 中的類🄵), 980
Timer (timeit 中的類🄵), 1887
timerfd_create() (於 os 模組中), 712
timerfd_gettime() (於 os 模組中), 714
timerfd_gettime_ns() (於 os 模組中), 714
timerfd_settime() (於 os 模組中), 713
timerfd_settime_ns() (於 os 模組中), 714
TimerHandle (asyncio 中的類🄵), 1139
times() (於 os 模組中), 725
TIMESTAMP (py_compile.PycInvalidationMode 的屬性), 2146
timestamp() (datetime.datetime 的方法), 231
timetuple() (datetime.date 的方法), 222
timetuple() (datetime.datetime 的方法), 231
timetz() (datetime.datetime 的方法), 230
timezone (datetime 中的類🄵), 246
timezone (於 time 模組中), 759
--timing
    trace 命令列選項, 1891
title() (bytearray 的方法), 79
title() (bytes 的方法), 79
title() (str 的方法), 62
title() (於 turtle 模組中), 1659
Tk, 1579
Tk (tkinter 中的類🄵), 1581
tk (tkinter.Tk 的屬性), 1581
Tk 選項資料型🄵, 1588
Tkinter, 1579
tkinter
    module, 1579
tkinter.colorchooser
    module, 1591
tkinter.commondialog
    module, 1595
tkinter.dnd
    module, 1598
tkinter.filedialog
    module, 1593
tkinter.font
    module, 1591
tkinter.messagebox
    module, 1595
tkinter.scrolledtext
    module, 1597
tkinter.simpledialog
    module, 1592
tkinter.ttk
    module, 1599
TLS, 1202
--tls-cert
    http.server 命令列選項, 1518
--tls-key
    http.server 命令列選項, 1518
--tls-password-file
    http.server 命令列選項, 1518
TLSv1 (ssl.TLSVersion 的屬性), 1214
TLSv1_1 (ssl.TLSVersion 的屬性), 1214
TLSv1_2 (ssl.TLSVersion 的屬性), 1214
TLSv1_3 (ssl.TLSVersion 的屬性), 1214
TLSVersion (ssl 中的類🄵), 1214
tm_gmtoff (time.struct_time 的屬性), 755
tm_hour (time.struct_time 的屬性), 755
tm_isdst (time.struct_time 的屬性), 755
tm_mday (time.struct_time 的屬性), 755
tm_min (time.struct_time 的屬性), 755
tm_mon (time.struct_time 的屬性), 755
tm_sec (time.struct_time 的屬性), 755
tm_wday (time.struct_time 的屬性), 755
tm_yday (time.struct_time 的屬性), 755
tm_year (time.struct_time 的屬性), 755
tm_zone (time.struct_time 的屬性), 755
TMP, 486
TMPDIR, 486
TO_BOOL (opcode), 2158
to_bytes() (int 的方法), 41
to_eng_string() (decimal.Context 的方法), 371
to_eng_string() (decimal.Decimal 的方法), 363
to_integral() (decimal.Decimal 的方法), 364
to_integral_exact() (decimal.Context 的方法), 371
to_integral_exact() (decimal.Decimal 的方法), 364
to_integral_value() (decimal.Decimal 的方法), 364
to_sci_string() (decimal.Context 的方法), 371
to_thread() (於 asyncio 模組中), 1091
ToASCII() (於 encodings.idna 模組中), 210
tobuf() (tarfile.TarInfo 的方法), 610
tobytes() (array.array 的方法), 296
tobytes() (memoryview 的方法), 84
today() (datetime.date 的類🄵方法), 219
today() (datetime.datetime 的類🄵方法), 225
tofile() (array.array 的方法), 296
tok_name (於 token 模組中), 2134
token, **2242**
    module, 2134
Token (contextvars 中的類🄵), 1068
token (shlex.shlex 的屬性), 2194
token_bytes() (於 secrets 模組中), 664
token_hex() (於 secrets 模組中), 664
token_urlsafe() (於 secrets 模組中), 664
TokenError, 2140
tokenize
    module, 2139
tokenize 命令列選項
    -e, 2141
    --exact, 2141
    -h, 2141
    --help, 2141
tokenize() (於 tokenize 模組中), 2139
tolist() (array.array 的方法), 296
tolist() (memoryview 的方法), 85

TOMLDecodeError, 645
tomllib
    module, 644
toordinal() (*datetime.date* 的方法), 222
toordinal() (*datetime.datetime* 的方法), 231
top() (*curses.panel.Panel* 的方法), 959
top() (*poplib.POP3* 的方法), 1482
top_panel() (於 *curses.panel* 模組中), 958
--top-level-directory
    unittest-discover 命令列選項, 1750
TopologicalSorter (*graphlib* 中的類F), 334
toprettyxml() (*xml.dom.minidom.Node* 的方法), 1396
toreadonly() (*memoryview* 的方法), 85
tostring() (於 *xml.etree.ElementTree* 模組中), 1375
tostringlist() (於 *xml.etree.ElementTree* 模組中), 1375
total() (*collections.Counter* 的方法), 267
total_changes (*sqlite3.Connection* 的屬性), 548
total_nframe (*tracemalloc.Traceback* 的屬性), 1902
total_ordering() (於 *functools* 模組中), 428
total_seconds() (*datetime.timedelta* 的方法), 218
touch() (*pathlib.Path* 的方法), 462
touchline() (*curses.window* 的方法), 942
touchwin() (*curses.window* 的方法), 942
tounicode() (*array.array* 的方法), 296
ToUnicode() (於 *encodings.idna* 模組中), 210
towards() (於 *turtle* 模組中), 1640
toxml() (*xml.dom.minidom.Node* 的方法), 1396
tparm() (於 *curses* 模組中), 934
trace
    module, 1890
--trace
    trace 命令列選項, 1891
Trace (*trace* 中的類F), 1892
Trace (*tracemalloc* 中的類F), 1901
trace function, 968, 1931, 1940
trace 命令列選項
    -C, 1891
    -c, 1891
    --count, 1891
    --coverdir, 1891
    -f, 1891
    --file, 1891
    -g, 1891
    --help, 1891
    --ignore-dir, 1892
    --ignore-module, 1892
    -l, 1891
    --listfuncs, 1891
    -m, 1891
    --missing, 1891
    --no-report, 1891
    -R, 1891
    -r, 1891
    --report, 1891
    -s, 1891
    --summary, 1891
    -T, 1891
    -t, 1891
    --timing, 1891
    --trace, 1891
    --trackcalls, 1891
    --version, 1891
trace() (於 *inspect* 模組中), 2031
trace_dispatch() (*bdb.Bdb* 的方法), 1861
traceback
    module, 2000
    object（物件）, 1924, 2000
Traceback (*inspect* 中的類F), 2029
Traceback (*tracemalloc* 中的類F), 1902
traceback (*tracemalloc.Statistic* 的屬性), 1901
traceback (*tracemalloc.StatisticDiff* 的屬性), 1901
traceback (*tracemalloc.Trace* 的屬性), 1902
traceback_limit (*tracemalloc.Snapshot* 的屬性), 1900
traceback_limit (*wsgiref.handlers.BaseHandler* 的屬性), 1432
TracebackException (*traceback* 中的類F), 2003
tracebacklimit (於 *sys* 模組中), 1945
TracebackType (*types* 中的類F), 307
tracemalloc
    module, 1893
tracer() (於 *turtle* 模組中), 1654
traces (*tracemalloc.Snapshot* 的屬性), 1901
--trackcalls
    trace 命令列選項, 1891
train_dict() (於 *compression.zstd* 模組中), 567
transfercmd() (*ftplib.FTP* 的方法), 1477
transient_internet() (於 *test.support.socket_helper* 模組中), 1846
translate() (*bytearray* 的方法), 73
translate() (*bytes* 的方法), 73
translate() (*str* 的方法), 62
translate() (於 *fnmatch* 模組中), 491
translate() (於 *glob* 模組中), 489
translation() (於 *gettext* 模組中), 1563
Transport (*asyncio* 中的類F), 1148
transport (*asyncio.StreamWriter* 的屬性), 1102
Transport Layer Security（傳輸層安全）, 1202
traps (*decimal.Context* 的屬性), 366
Traversable (*importlib.abc* 中的類F), 2067
Traversable (*importlib.resources.abc* 中的類F), 2082
TraversableResources (*importlib.abc* 中的類F), 2068
TraversableResources (*importlib.resources.abc* 中的類F), 2083
TreeBuilder (*xml.etree.ElementTree* 中的類F), 1381
Treeview (*tkinter.ttk* 中的類F), 1609
triangular() (於 *random* 模組中), 387
tries (*doctest.DocTestRunner* 的屬性), 1742
triple-quoted string（三引號F字串）, **2242**
True, 37, 45
true, 37
True (F建變數), 35

truediv() (於 operator 模組中), 437
trunc() (於 math 模組中), 343
trunc() (於 math 模組), 39
truncate() (io.IOBase 的方法), 738
truncate() (於 os 模組中), 707
truth() (於 operator 模組中), 436
truth (真)
    value, 37
try
    statement (陳述式), 109
Try (ast 中的類F), 2112
TryStar (ast 中的類F), 2112
TSTRING_END (於 token 模組中), 2135
TSTRING_MIDDLE (於 token 模組中), 2135
TSTRING_START (於 token 模組中), 2135
ttk, 1599
tty
    I/O control (I/O 控制), 2199
    module, 2200
ttyname() (於 os 模組中), 688
TUESDAY (於 calendar 模組中), 260
Tuple (ast 中的類F), 2100
tuple (F建類F), 50
Tuple (於 typing 模組中), 1715
tuple (元組)
    object (物件), 48, 50
turtle
    module, 1628
Turtle (turtle 中的類F), 1659
turtledemo
    module, 1663
turtles() (於 turtle 模組中), 1658
TurtleScreen (turtle 中的類F), 1659
turtlesize() (於 turtle 模組中), 1647
--type
    calendar 命令列選項, 263
type (ctypes.CField 的屬性), 852
type (optparse.Option 的屬性), 912
type (socket.socket 的屬性), 1198
type (tarfile.TarInfo 的屬性), 610
Type (typing 中的類F), 1715
type (urllib.request.Request 的屬性), 1441
type (F建類F), 30
type alias (型FF名), **2242**
type hint (型F提示), **2243**
TYPE_ALIAS (symtable.SymbolTableType 的屬性), 2130
type_check_only() (於 typing 模組中), 1710
TYPE_CHECKER (optparse.Option 的屬性), 922
TYPE_CHECKING (於 typing 模組中), 1714
type_comment (ast.arg 的屬性), 2122
type_comment (ast.Assign 的屬性), 2107
type_comment (ast.For 的屬性), 2110
type_comment (ast.FunctionDef 的屬性), 2121
type_comment (ast.With 的屬性), 2113
TYPE_COMMENT (於 token 模組中), 2135
TYPE_IGNORE (於 token 模組中), 2135
TYPE_PARAMETERS (symtable.SymbolTableType 的屬性), 2130
type_repr() (於 annotationlib 模組中), 2040
TYPE_VARIABLE (symtable.SymbolTableType 的屬性), 2130
typeahead() (於 curses 模組中), 935
TypeAlias (ast 中的類F), 2109
TypeAlias (於 typing 模組中), 1681
TypeAliasType (typing 中的類F), 1697
typecode (array.array 的屬性), 294
typecodes (於 array 模組中), 294
TYPED_ACTIONS (optparse.Option 的屬性), 923
typed_subpart_iterator() (於 email.iterators 模組中), 1318
TypedDict (typing 中的類F), 1701
TypeError, 115
TypeGuard (於 typing 模組中), 1688
TypeIgnore (ast 中的類F), 2119
TypeIs (於 typing 模組中), 1687
types
    module, 303
TYPES (optparse.Option 的屬性), 922
types_map (mimetypes.MimeTypes 的屬性), 1350
types_map (於 mimetypes 模組中), 1349
types_map_inv (mimetypes.MimeTypes 的屬性), 1350
TypeVar (ast 中的類F), 2119
TypeVar (typing 中的類F), 1690
TypeVarTuple (ast 中的類F), 2120
TypeVarTuple (typing 中的類F), 1693
type (型F), **2242**
    Boolean (布林值), 9
    built-in function (F建函式), 104
    built-in (F建), 37
    immutable (不可變) sequence (序列), 48
    mutable (可變) sequence (序列), 48
    object (物件), 30
    operations on (操作於) dictionary (字典), 91
    operations on (操作於) integer (整數), 40
    operations on (操作於) list (串列), 48
    operations on (操作於) mapping (對映), 91
    operations on (操作於) numeric (數值), 39
    operations on (操作於) sequence (序列), 46, 48
    union (聯集), 101
    模組, 104
typing
    module, 1667
TZ, 756, 757
tzinfo (datetime 中的類F), 239
tzinfo (datetime.datetime 的屬性), 228
tzinfo (datetime.time 的屬性), 236
tzname (於 time 模組中), 759
tzname() (datetime.datetime 的方法), 231

tzname() (*datetime.time* 的方法), 239
tzname() (*datetime.timezone* 的方法), 246
tzname() (*datetime.tzinfo* 的方法), 240
TZPATH (於 *zoneinfo* 模組中), 255
tzset() (於 *time* 模組中), 756

U

-u
    timeit 命令列選項, 1888
    uuid 命令列選項, 1501
U (於 *re* 模組中), 147
UAdd (*ast* 中的類🄵), 2102
ucd_3_2_0 (於 *unicodedata* 模組中), 178
udata (*select.kevent* 的屬性), 1243
UDPServer (*socketserver* 中的類🄵), 1503
UF_APPEND (於 *stat* 模組中), 479
UF_COMPRESSED (於 *stat* 模組中), 479
UF_DATAVAULT (於 *stat* 模組中), 479
UF_HIDDEN (於 *stat* 模組中), 479
UF_IMMUTABLE (於 *stat* 模組中), 479
UF_NODUMP (於 *stat* 模組中), 479
UF_NOUNLINK (於 *stat* 模組中), 479
UF_OPAQUE (於 *stat* 模組中), 479
UF_SETTABLE (於 *stat* 模組中), 479
UF_TRACKED (於 *stat* 模組中), 479
UID (*plistlib* 中的類🄵), 648
uid (*tarfile.TarInfo* 的屬性), 611
uid() (*imaplib.IMAP4* 的方法), 1489
uidl() (*poplib.POP3* 的方法), 1482
ulp() (於 *math* 模組中), 344
umask() (於 *os* 模組中), 675
unalias (*pdb command*), 1876
uname (*tarfile.TarInfo* 的屬性), 611
uname() (於 *os* 模組中), 676
uname() (於 *platform* 模組中), 806
UNARY_INVERT (*opcode*), 2158
UNARY_NEGATIVE (*opcode*), 2158
UNARY_NOT (*opcode*), 2158
UnaryOp (*ast* 中的類🄵), 2102
UnboundLocalError, 115
unbuffered I/O (非緩衝 I/O), 24
UNC paths (UNC 路徑)
    以及 os.makedirs(), 696
uncancel() (*asyncio.Task* 的方法), 1097
UNCHECKED_HASH (*py_compile.PycInvalidationMode* 的屬性), 2146
unconsumed_tail (*zlib.Decompress* 的屬性), 576
unctrl() (於 *curses* 模組中), 935
unctrl() (於 *curses.ascii* 模組中), 958
Underflow (*decimal* 中的類🄵), 373
undisplay (*pdb command*), 1875
undo() (於 *turtle* 模組中), 1639
undobufferentries() (於 *turtle* 模組中), 1651
undoc_header (*cmd.Cmd* 的屬性), 961
unescape() (於 *html* 模組中), 1359
unescape() (於 *xml.sax.saxutils* 模組中), 1407
UnexpectedException, 1745
unexpectedSuccesses (*unittest.TestResult* 的屬性), 1770
unfreeze() (於 *gc* 模組中), 2015
unget_wch() (於 *curses* 模組中), 935
ungetch() (於 *curses* 模組中), 935
ungetch() (於 *msvcrt* 模組中), 2178
ungetmouse() (於 *curses* 模組中), 935
ungetwch() (於 *msvcrt* 模組中), 2178
unhexlify() (於 *binascii* 模組中), 1357
Unicode, 176, 194
    database (資料庫), 176
UNICODE (於 *re* 模組中), 147
unicode_literals (於 *__future__* 模組中), 2011
unicodedata
    module, 176
UnicodeDecodeError, 116
UnicodeEncodeError, 115
UnicodeError, 115
UnicodeTranslateError, 116
UnicodeWarning, 118
unidata_version (於 *unicodedata* 模組中), 178
unified_diff() (於 *difflib* 模組中), 164
uniform() (於 *random* 模組中), 387
UnimplementedFileMode, 1468
Union (*ctypes* 中的類🄵), 849
Union (*typing* 中的類🄵), 1681
union() (*frozenset* 的方法), 90
UnionType (*types* 中的類🄵), 307
Union (聯合)
    object (物件), 101
union (聯集)
    type (型🄵), 101
UNIQUE (*enum.EnumCheck* 的屬性), 329
unique() (於 *enum* 模組中), 333
--unit
    timeit 命令列選項, 1888
unittest
    module, 1747
unittest 命令列選項
    -b, 1749
    --buffer, 1749
    -c, 1749
    --catch, 1749
    --durations, 1749
    -f, 1749
    --failfast, 1749
    -k, 1749
    --locals, 1749
unittest-discover 命令列選項
    -p, 1750
    --pattern, 1750
    -s, 1750
    --start-directory, 1750
    -t, 1750
    --top-level-directory, 1750
    -v, 1750
    --verbose, 1750
unittest.mock

module, 1777
universal newlines
bytearray.splitlines 方法, 78
bytes.splitlines 方法, 78
csv.reader 函式, 619
importlib.abc.InspectLoader.get_source 方法, 2064
io.IncrementalNewlineDecoder 類🄵, 746
io.TextIOWrapper 類🄵, 744
open() 🄵建函式, 23
str.splitlines 方法, 61
subprocess 模組, 1047
universal newlines（通用🄵行字元）, **2243**
UNIX
file control（檔案控制）, 2203
I/O control（I/O 控制）, 2203
unix_dialect (*csv* 中的類🄵), 622
unix_shell (於 *test.support* 模組中), 1837
UnixDatagramServer (*socketserver* 中的類🄵), 1503
UnixStreamServer (*socketserver* 中的類🄵), 1503
unknown (*uuid.SafeUUID* 的屬性), 1497
unknown_decl() (*html.parser.HTMLParser* 的方法), 1362
unknown_open() (*urllib.request.BaseHandler* 的方法), 1444
unknown_open() (*urllib.request.UnknownHandler* 的方法), 1447
UnknownHandler (*urllib.request* 中的類🄵), 1440
UnknownProtocol, 1468
UnknownTransferEncoding, 1468
unlink() (*multiprocessing.shared_memory.SharedMemory* 的方法), 1027
unlink() (*pathlib.Path* 的方法), 465
unlink() (於 *os* 模組中), 707
unlink() (於 *test.support.os_helper* 模組中), 1850
unlink() (*xml.dom.minidom.Node* 的方法), 1395
unload() (於 *test.support.import_helper* 模組中), 1851
unlock() (*mailbox.Babyl* 的方法), 1338
unlock() (*mailbox.Mailbox* 的方法), 1332
unlock() (*mailbox.Maildir* 的方法), 1335
unlock() (*mailbox.mbox* 的方法), 1335
unlock() (*mailbox.MH* 的方法), 1337
unlock() (*mailbox.MMDF* 的方法), 1338
unlockpt() (於 *os* 模組中), 688
UNNAMED_SECTION (於 *configparser* 模組中), 642
UnnamedSectionDisabledError, 644
Unpack (於 *typing* 模組中), 1689
unpack() (*struct.Struct* 的方法), 193
unpack() (於 *struct* 模組中), 188
unpack_archive() (於 *shutil* 模組中), 500
UNPACK_EX (*opcode*), 2162
unpack_from() (*struct.Struct* 的方法), 193
unpack_from() (於 *struct* 模組中), 188
UNPACK_SEQUENCE (*opcode*), 2162
unparse() (於 *ast* 模組中), 2125
unparsedEntityDecl() (*xml.sax.handler.DTDHandler* 的方法), 1405
UnparsedEntityDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
Unpickler (*pickle* 中的類🄵), 509
UnpicklingError, 508
unquote() (於 *email.utils* 模組中), 1316
unquote() (於 *urllib.parse* 模組中), 1459
unquote_plus() (於 *urllib.parse* 模組中), 1460
unquote_to_bytes() (於 *urllib.parse* 模組中), 1460
unraisablehook() (於 *sys* 模組中), 1945
unregister() (*select.devpoll* 的方法), 1239
unregister() (*select.epoll* 的方法), 1240
unregister() (*selectors.BaseSelector* 的方法), 1245
unregister() (*select.poll* 的方法), 1241
unregister() (於 *atexit* 模組中), 1999
unregister() (於 *codecs* 模組中), 195
unregister() (於 *faulthandler* 模組中), 1867
unregister_archive_format() (於 *shutil* 模組中), 500
unregister_dialect() (於 *csv* 模組中), 620
unregister_unpack_format() (於 *shutil* 模組中), 501
unsafe (*uuid.SafeUUID* 的屬性), 1497
unselect() (*imaplib.IMAP4* 的方法), 1489
unset() (*test.support.os_helper.EnvironmentVarGuard* 的方法), 1849
unsetenv() (於 *os* 模組中), 676
unshare() (於 *os* 模組中), 676
UnstructuredHeader (*email.headerregistry* 中的類🄵), 1287
unsubscribe() (*imaplib.IMAP4* 的方法), 1489
UnsupportedOperation, 445, 736
until (*pdb command*), 1874
untokenize() (於 *tokenize* 模組中), 2140
untouchwin() (*curses.window* 的方法), 942
unused_data (*bz2.BZ2Decompressor* 的屬性), 584
unused_data (*compression.zstd.ZstdDecompressor* 的屬性), 567
unused_data (*lzma.LZMADecompressor* 的屬性), 588
unused_data (*zlib.Decompress* 的屬性), 576
unverifiable (*urllib.request.Request* 的屬性), 1441
unwrap() (*ssl.SSLSocket* 的方法), 1217
unwrap() (於 *inspect* 模組中), 2028
unwrap() (於 *urllib.parse* 模組中), 1457
up (*pdb command*), 1872
up() (於 *turtle* 模組中), 1642
update() (*collections.Counter* 的方法), 267
update() (*dict* 的方法), 94
update() (*frozenset* 的方法), 91
update() (*hashlib.hash* 的方法), 653
update() (*hmac.HMAC* 的方法), 662
update() (*http.cookies.Morsel* 的方法), 1521
update() (*mailbox.Mailbox* 的方法), 1332
update() (*mailbox.Maildir* 的方法), 1334
update() (*trace.CoverageResults* 的方法), 1892
update() (於 *turtle* 模組中), 1654
update_abstractmethods() (於 *abc* 模組中), 1998

update_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 的方法), 1446
update_lines_cols() (於 *curses* 模組中), 935
update_panels() (於 *curses.panel* 模組中), 958
update_visible() (*mailbox.BabylMessage* 的方法), 1344
update_wrapper() (於 *functools* 模組中), 434
--upgrade
    venv 命令列選項, 1906
upgrade_dependencies() (*venv.EnvBuilder* 的方法), 1909
--upgrade-deps
    venv 命令列選項, 1906
upper() (*bytearray* 的方法), 79
upper() (*bytes* 的方法), 79
upper() (*str* 的方法), 62
urandom() (於 *os* 模組中), 733
url (*http.client.HTTPResponse* 的屬性), 1472
url (*urllib.error.HTTPError* 的屬性), 1461
url (*urllib.response.addinfourl* 的屬性), 1452
url (*xmlrpc.client.ProtocolError* 的屬性), 1535
url2pathname() (於 *urllib.request* 模組中), 1437
urlcleanup() (於 *urllib.request* 模組中), 1451
urldefrag() (於 *urllib.parse* 模組中), 1457
urlencode() (於 *urllib.parse* 模組中), 1460
URLError, 1461
urljoin() (於 *urllib.parse* 模組中), 1456
urllib
    module, 1435
urllib.error
    module, 1461
urllib.parse
    module, 1452
urllib.request
    module, 1435
    module（模組）, 1466
urllib.response
    module, 1452
urllib.robotparser
    module, 1462
urlopen() (於 *urllib.request* 模組中), 1435
urlparse() (於 *urllib.parse* 模組中), 1453
urlretrieve() (於 *urllib.request* 模組中), 1450
urlsafe_b64decode() (於 *base64* 模組中), 1353
urlsafe_b64encode() (於 *base64* 模組中), 1353
urlsplit() (於 *urllib.parse* 模組中), 1455
urlunparse() (於 *urllib.parse* 模組中), 1455
urlunsplit() (於 *urllib.parse* 模組中), 1456
URL（統一資源定位器）, 1452, 1462, 1511
    parsing（剖析）, 1452
    relative（相對）, 1452
urn (*uuid.UUID* 的屬性), 1499
US (於 *curses.ascii* 模組中), 956
use_default_colors() (於 *curses* 模組中), 935
use_env() (於 *curses* 模組中), 935
use_rawinput (*cmd.Cmd* 的屬性), 961
use_tool_id() (於 *sys.monitoring* 模組中), 1947
UseForeignDTD() (*xml.parsers.expat.xmlparser* 的方法), 1413
USER, 926
--user
    ensurepip 命令列選項, 1904
user() (*poplib.POP3* 的方法), 1481
USER_BASE (於 *site* 模組中), 2045
user_call() (*bdb.Bdb* 的方法), 1862
user_exception() (*bdb.Bdb* 的方法), 1862
user_line() (*bdb.Bdb* 的方法), 1862
user_return() (*bdb.Bdb* 的方法), 1862
USER_SITE (於 *site* 模組中), 2045
--user-base
    site 命令列選項, 2046
usercustomize
    module, 2044
UserDict (*collections* 中的類🄵), 279
UserList (*collections* 中的類🄵), 279
USERNAME, 470, 672, 926
username (*email.headerregistry.Address* 的屬性), 1291
USERPROFILE, 470
userptr() (*curses.panel.Panel* 的方法), 959
--user-site
    site 命令列選項, 2046
UserString (*collections* 中的類🄵), 279
UserWarning, 117
user（使用者）
    effective id, 671
    id, 673
    id, setting（設定）, 675
USTAR_FORMAT (於 *tarfile* 模組中), 605
USub (*ast* 中的類🄵), 2102
UTC, 748
utc (*datetime.timezone* 的屬性), 246
UTC (於 *datetime* 模組中), 214
utcfromtimestamp() (*datetime.datetime* 的類🄵方法), 226
utcnow() (*datetime.datetime* 的類🄵方法), 225
utcoffset() (*datetime.datetime* 的方法), 230
utcoffset() (*datetime.time* 的方法), 238
utcoffset() (*datetime.timezone* 的方法), 246
utcoffset() (*datetime.tzinfo* 的方法), 239
utctimetuple() (*datetime.datetime* 的方法), 231
utf8 (*email.policy.EmailPolicy* 的屬性), 1282
utf8() (*poplib.POP3* 的方法), 1482
utf8_enabled (*imaplib.IMAP4* 的屬性), 1490
utf8_mode (*sys.flags* 的屬性), 1926
utime() (於 *os* 模組中), 707
uu
    module, 2223
uuid
    module, 1497
--uuid
    uuid 命令列選項, 1501
UUID (*uuid* 中的類🄵), 1497
uuid 命令列選項
    -C, 1501
    --count, 1501

-h, 1501
--help, 1501
-N, 1501
-n, 1501
--name, 1501
--namespace, 1501
-u, 1501
--uuid, 1501
uuid1() (於 *uuid* 模組中), 1499
uuid3() (於 *uuid* 模組中), 1499
uuid4() (於 *uuid* 模組中), 1499
uuid5() (於 *uuid* 模組中), 1499
uuid6() (於 *uuid* 模組中), 1499
uuid7() (於 *uuid* 模組中), 1499
uuid8() (於 *uuid* 模組中), 1500

V

-v
doctest 命令列選項, 1727
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令列選項, 554
tarfile 命令列選項, 616
timeit 命令列選項, 1888
unittest-discover 命令列選項, 1750
v4_int_to_packed() (於 *ipaddress* 模組中), 1556
v6_int_to_packed() (於 *ipaddress* 模組中), 1556
valid_signals() (於 *signal* 模組中), 1251
validator() (於 *wsgiref.validate* 模組中), 1430
value
truth（真）, 37
VALUE (*annotationlib.Format* 的屬性), 2036
value (*ctypes._SimpleCData* 的屬性), 846
value (*enum.Enum* 的屬性), 323
value (*http.cookiejar.Cookie* 的屬性), 1528
value (*http.cookies.Morsel* 的屬性), 1520
value (*StopIteration* 的屬性), 114
value (*string.templatelib.Interpolation* 的屬性), 138
value (*xml.dom.Attr* 的屬性), 1391
Value() (*multiprocessing.managers.SyncManager* 的方法), 1006
Value() (於 *multiprocessing* 模組中), 1001
Value() (於 *multiprocessing.sharedctypes* 模組中), 1003
value_decode() (*http.cookies.BaseCookie* 的方法), 1519
value_encode() (*http.cookies.BaseCookie* 的方法), 1519
VALUE_WITH_FAKE_GLOBALS (*annotationlib.Format* 的屬性), 2036
ValueError, 116
valuerefs() (*weakref.WeakValueDictionary* 的方法), 298
values
Boolean（布林）, 45
Values (*optparse* 中的類🄵), 911
values (*string.templatelib.Template* 的屬性), 136
values() (*contextvars.Context* 的方法), 1070
values() (*dict* 的方法), 94
values() (*email.message.EmailMessage* 的方法), 1266
values() (*email.message.Message* 的方法), 1304
values() (*mailbox.Mailbox* 的方法), 1331
values() (*types.MappingProxyType* 的方法), 308
ValuesView (*collections.abc* 中的類🄵), 284
ValuesView (*typing* 中的類🄵), 1717
var (*contextvars.Token* 的屬性), 1069
variable annotation（變數🄵釋）, **2243**
variance (*statistics.NormalDist* 的屬性), 404
variance() (於 *statistics* 模組中), 400
variant (*uuid.UUID* 的屬性), 1499
vars()
built-in function, 30
vbar (*tkinter.scrolledtext.ScrolledText* 的屬性), 1598
VBAR (於 *token* 模組中), 2136
VBAREQUAL (於 *token* 模組中), 2138
VC_ASSEMBLY_PUBLICKEYTOKEN (於 *msvcrt* 模組中), 2179
Vec2D (*turtle* 中的類🄵), 1660
venv
module, 1905
venv 命令列選項
--clear, 1906
--copies, 1906
ENV_DIR, 1906
--prompt, 1906
--symlinks, 1906
--system-site-packages, 1906
--upgrade, 1906
--upgrade-deps, 1906
--without-pip, 1906
--without-scm-ignore-files, 1906
--verbose
doctest 命令列選項, 1727
tarfile 命令列選項, 616
timeit 命令列選項, 1888
unittest-discover 命令列選項, 1750
verbose (*sys.flags* 的屬性), 1926
VERBOSE (於 *re* 模組中), 148
verbose (於 *tabnanny* 模組中), 2143
verbose (於 *test.support* 模組中), 1837
verify() (*smtplib.SMTP* 的方法), 1493
verify() (於 *enum* 模組中), 333
VERIFY_ALLOW_PROXY_CERTS (於 *ssl* 模組中), 1208
verify_client_post_handshake() (*ssl.SSLSocket* 的方法), 1217
verify_code (*ssl.SSLCertVerificationError* 的屬性), 1205
VERIFY_CRL_CHECK_CHAIN (於 *ssl* 模組中), 1208
VERIFY_CRL_CHECK_LEAF (於 *ssl* 模組中), 1208
VERIFY_DEFAULT (於 *ssl* 模組中), 1208
verify_flags (*ssl.SSLContext* 的屬性), 1226
verify_generated_headers (*email.policy.Policy* 的屬性), 1280
verify_message (*ssl.SSLCertVerificationError* 的屬性), 1205
verify_mode (*ssl.SSLContext* 的屬性), 1226

verify_request() (*socketserver.BaseServer* 的方法), 1507
VERIFY_X509_PARTIAL_CHAIN (於 *ssl* 模組中), 1209
VERIFY_X509_STRICT (於 *ssl* 模組中), 1208
VERIFY_X509_TRUSTED_FIRST (於 *ssl* 模組中), 1208
VerifyFlags (*ssl* 中的類🄵), 1209
VerifyMode (*ssl* 中的類🄵), 1208
--version
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令列選項, 554
    trace 命令列選項, 1891
version (*email.headerregistry.MIMEVersionHeader* 的屬性), 1289
version (*http.client.HTTPResponse* 的屬性), 1472
version (*http.cookiejar.Cookie* 的屬性), 1528
version (*http.cookies.Morsel* 的屬性), 1520
version (*ipaddress.IPv4Address* 的屬性), 1544
version (*ipaddress.IPv4Network* 的屬性), 1550
version (*ipaddress.IPv6Address* 的屬性), 1547
version (*ipaddress.IPv6Network* 的屬性), 1552
version (*sys.thread_info* 的屬性), 1945
version (*uuid.UUID* 的屬性), 1499
version (於 *curses* 模組中), 942
version (於 *marshal* 模組中), 525
version (於 *sys* 模組中), 1945
version() (*ssl.SSLSocket* 的方法), 1218
version() (於 *ensurepip* 模組中), 1904
version() (於 *importlib.metadata* 模組中), 2087
version() (於 *platform* 模組中), 805
version_info (於 *sys* 模組中), 1946
version_string() (*http.server.BaseHTTPRequestHandler* 的方法), 1515
vformat() (*string.Formatter* 的方法), 124
基准量測 (*Benchmarking*), 1886
virtual environment (🄵擬環境), **2243**
virtual machine (🄵擬機器), **2243**
virtual 🄵擬
    Environments (環境), 1905
visit() (*ast.NodeVisitor* 的方法), 2126
visit_Constant() (*ast.NodeVisitor* 的方法), 2127
安全雜🄵演算法、SHA1、SHA2、SHA224、SHA256、SHA384、SHA512、SHA3、Shake、Blake2, 651
vline() (*curses.window* 的方法), 942
voidcmd() (*ftplib.FTP* 的方法), 1476
volume (*zipfile.ZipInfo* 的屬性), 600
vonmisesvariate() (於 *random* 模組中), 388
VT (於 *curses.ascii* 模組中), 955

W

-w
    calendar 命令列選項, 263
W_OK (於 *os* 模組中), 691
性能表現, 1886
wait() (*asyncio.Barrier* 的方法), 1110
wait() (*asyncio.Condition* 的方法), 1108
wait() (*asyncio.Event* 的方法), 1107
wait() (*asyncio.subprocess.Process* 的方法), 1113
wait() (*multiprocessing.pool.AsyncResult* 的方法), 1012
wait() (*subprocess.Popen* 的方法), 1053
wait() (*threading.Barrier* 的方法), 981
wait() (*threading.Condition* 的方法), 977
wait() (*threading.Event* 的方法), 980
wait() (於 *asyncio* 模組中), 1090
wait() (於 *concurrent.futures* 模組中), 1038
wait() (於 *multiprocessing.connection* 模組中), 1014
wait() (於 *os* 模組中), 725
wait3() (於 *os* 模組中), 726
wait4() (於 *os* 模組中), 726
wait_closed() (*asyncio.Server* 的方法), 1140
wait_closed() (*asyncio.StreamWriter* 的方法), 1102
wait_for() (*asyncio.Condition* 的方法), 1108
wait_for() (*threading.Condition* 的方法), 977
wait_for() (於 *asyncio* 模組中), 1089
wait_process() (於 *test.support* 模組中), 1841
wait_threads_exit() (於 *test.support.threading_helper* 模組中), 1848
wait_until_any_call_with() (*unittest.mock.ThreadingMock* 的方法), 1792
wait_until_called() (*unittest.mock.ThreadingMock* 的方法), 1792
waitid() (於 *os* 模組中), 725
waitpid() (於 *os* 模組中), 726
waitstatus_to_exitcode() (於 *os* 模組中), 728
walk() (*email.message.EmailMessage* 的方法), 1269
walk() (*email.message.Message* 的方法), 1307
walk() (*pathlib.Path* 的方法), 461
walk() (於 *ast* 模組中), 2126
walk() (於 *os* 模組中), 708
walk_packages() (於 *pkgutil* 模組中), 2054
walk_stack() (於 *traceback* 模組中), 2002
walk_tb() (於 *traceback* 模組中), 2003
walrus operator (海象運算子), **2243**
want (*doctest.Example* 的屬性), 1739
warn() (於 *warnings* 模組中), 1966
warn_default_encoding (*sys.flags* 的屬性), 1926
warn_explicit() (於 *warnings* 模組中), 1967
Warning, 117, 552
WARNING (於 *logging* 模組中), 766
WARNING (於 *tkinter.messagebox* 模組中), 1597
warning() (*logging.Logger* 的方法), 764
warning() (於 *logging* 模組中), 774
warning() (*xml.sax.handler.ErrorHandler* 的方法), 1406
warnings
    module, 1962
warnings (警告), 1962
WarningsRecorder (*test.support.warnings_helper* 中的類🄵), 1853
warnoptions (於 *sys* 模組中), 1946
wasSuccessful() (*unittest.TestResult* 的方法), 1771

WatchedFileHandler (*logging.handlers* 中的類🄵), 792
wave
    module, 1557
Wave_read (*wave* 中的類🄵), 1558
Wave_write (*wave* 中的類🄵), 1559
WCONTINUED (於 *os* 模組中), 727
WCOREDUMP() (於 *os* 模組中), 728
WeakKeyDictionary (*weakref* 中的類🄵), 298
WeakMethod (*weakref* 中的類🄵), 298
weakref
    module, 296
WeakSet (*weakref* 中的類🄵), 298
WeakValueDictionary (*weakref* 中的類🄵), 298
webbrowser
    module, 1423
webbrowser 命令列選項
    -n, 1423
    --new-tab, 1423
    --new-window, 1423
    -t, 1423
WEDNESDAY (於 *calendar* 模組中), 260
weekday (*calendar.IllegalWeekdayError* 的屬性), 261
weekday() (*datetime.date* 的方法), 222
weekday() (*datetime.datetime* 的方法), 232
weekday() (於 *calendar* 模組中), 259
weekheader() (於 *calendar* 模組中), 259
weibullvariate() (於 *random* 模組中), 388
WEXITED (於 *os* 模組中), 727
WEXITSTATUS() (於 *os* 模組中), 729
wfile (*http.server.BaseHTTPRequestHandler* 的屬性), 1513
wfile (*socketserver.DatagramRequestHandler* 的屬性), 1507
whatis (*pdb command*), 1874
when() (*asyncio.Timeout* 的方法), 1089
when() (*asyncio.TimerHandle* 的方法), 1139
whence (*concurrent.interpreters.Interpreter* 的屬性), 1042
where (*pdb command*), 1872
which() (於 *shutil* 模組中), 497
whichdb() (於 *dbm* 模組中), 526
while
    statement（陳述式）, 37
While (*ast* 中的類🄵), 2111
whitespace (*shlex.shlex* 的屬性), 2194
whitespace (於 *string* 模組中), 124
whitespace_split (*shlex.shlex* 的屬性), 2194
Widget (*tkinter.ttk* 中的類🄵), 1601
--width
    calendar 命令列選項, 263
width (*sys.hash_info* 的屬性), 1932
width (*textwrap.TextWrapper* 的屬性), 175
width() (於 *turtle* 模組中), 1642
標准函式庫, **2241**
WIFCONTINUED() (於 *os* 模組中), 728
WIFEXITED() (於 *os* 模組中), 729
WIFSIGNALED() (於 *os* 模組中), 729
WIFSTOPPED() (於 *os* 模組中), 729
模組
    array（陣列）, 66
    math, 39
    re, 54
    type（型🄵）, 104
win32_code_page_search_function() (於 *encodings* 模組中), 210
win32_edition() (於 *platform* 模組中), 806
win32_is_iot() (於 *platform* 模組中), 806
win32_ver() (於 *platform* 模組中), 806
WinDLL (*ctypes* 中的類🄵), 836
window (*curses* 中的類🄵), 936
window() (*curses.panel.Panel* 的方法), 959
window_height() (於 *turtle* 模組中), 1658
window_log (*compression.zstd.CompressionParameter* 的屬性), 569
window_log_max (*compression.zstd.DecompressionParameter* 的屬性), 572
window_width() (於 *turtle* 模組中), 1658
Windows ini file（Windows ini 檔案）, 626
WindowsError, 116
WindowsPath (*pathlib* 中的類🄵), 454
WindowsProactorEventLoopPolicy (*asyncio* 中的類🄵), 1161
WindowsRegistryFinder (*importlib.machinery* 中的類🄵), 2069
WindowsSelectorEventLoopPolicy (*asyncio* 中的類🄵), 1161
winerror (*OSError* 的屬性), 113
WinError() (於 *ctypes* 模組中), 845
WINFUNCTYPE() (於 *ctypes* 模組中), 840
winreg
    module, 2180
WinSock, 1238
winsound
    module, 2188
winver (於 *sys* 模組中), 1946
With (*ast* 中的類🄵), 2113
WITH_EXCEPT_START (*opcode*), 2161
with_hostmask (*ipaddress.IPv4Interface* 的屬性), 1555
with_hostmask (*ipaddress.IPv4Network* 的屬性), 1550
with_hostmask (*ipaddress.IPv6Interface* 的屬性), 1555
with_hostmask (*ipaddress.IPv6Network* 的屬性), 1553
with_name() (*pathlib.PurePath* 的方法), 452
with_netmask (*ipaddress.IPv4Interface* 的屬性), 1555
with_netmask (*ipaddress.IPv4Network* 的屬性), 1550
with_netmask (*ipaddress.IPv6Interface* 的屬性), 1555
with_netmask (*ipaddress.IPv6Network* 的屬性), 1553
with_prefixlen (*ipaddress.IPv4Interface* 的屬性), 1555
with_prefixlen (*ipaddress.IPv4Network* 的屬性), 1550

with_prefixlen (*ipaddress.IPv6Interface* 的屬性), 1555
with_prefixlen (*ipaddress.IPv6Network* 的屬性), 1553
with_pymalloc() (於 *test.support* 模組中), 1839
with_segments() (*pathlib.PurePath* 的方法), 453
with_statement (於 *__future__* 模組中), 2011
with_stem() (*pathlib.PurePath* 的方法), 453
with_suffix() (*pathlib.PurePath* 的方法), 453
with_traceback() (*BaseException* 的方法), 110
withitem (*ast* 中的類F), 2113
--without-pip
    venv 命令列選項, 1906
--without-scm-ignore-files
    venv 命令列選項, 1906
WNOHANG (於 *os* 模組中), 727
WNOWAIT (於 *os* 模組中), 727
wordchars (*shlex.shlex* 的屬性), 2194
World Wide Web (全球資訊網), 1423, 1452, 1462
wrap() (*textwrap.TextWrapper* 的方法), 176
wrap() (於 *textwrap* 模組中), 173
wrap_bio() (*ssl.SSLContext* 的方法), 1224
wrap_future() (於 *asyncio* 模組中), 1144
wrap_socket() (*ssl.SSLContext* 的方法), 1223
wrapper() (於 *curses* 模組中), 935
WrapperDescriptorType (於 *types* 模組中), 306
wraps() (於 *functools* 模組中), 434
WRITABLE (*inspect.BufferFlags* 的屬性), 2034
WRITABLE (於 *_tkinter* 模組中), 1591
writable() (*bz2.BZ2File* 的方法), 582
writable() (*io.IOBase* 的方法), 738
WRITE (*inspect.BufferFlags* 的屬性), 2035
write() (*asyncio.StreamWriter* 的方法), 1101
write() (*asyncio.WriteTransport* 的方法), 1150
write() (*codecs.StreamWriter* 的方法), 201
write() (*code.InteractiveInterpreter* 的方法), 2048
write() (*configparser.ConfigParser* 的方法), 641
write() (*email.generator.BytesGenerator* 的方法), 1276
write() (*email.generator.Generator* 的方法), 1277
write() (*io.BufferedIOBase* 的方法), 740
write() (*io.BufferedWriter* 的方法), 743
write() (*io.RawIOBase* 的方法), 739
write() (*io.TextIOBase* 的方法), 744
write() (*io.Writer* 的方法), 747
write() (*mmap.mmap* 的方法), 1259
write() (*sqlite3.Blob* 的方法), 551
write() (*ssl.MemoryBIO* 的方法), 1234
write() (*ssl.SSLSocket* 的方法), 1215
write() (於 *os* 模組中), 688
write() (於 *turtle* 模組中), 1646
write() (*xml.etree.ElementTree.ElementTree* 的方法), 1380
write() (*zipfile.ZipFile* 的方法), 595
write_byte() (*mmap.mmap* 的方法), 1259
write_bytes() (*pathlib.Path* 的方法), 460
write_docstringdict() (於 *turtle* 模組中), 1662
write_eof() (*asyncio.StreamWriter* 的方法), 1102
write_eof() (*asyncio.WriteTransport* 的方法), 1150
write_eof() (*ssl.MemoryBIO* 的方法), 1234
write_history_file() (於 *readline* 模組中), 181
write_results() (*trace.CoverageResults* 的方法), 1892
write_text() (*pathlib.Path* 的方法), 460
write_through (*io.TextIOWrapper* 的屬性), 745
writeframes() (*wave.Wave_write* 的方法), 1559
writeframesraw() (*wave.Wave_write* 的方法), 1559
writeheader() (*csv.DictWriter* 的方法), 625
writelines() (*asyncio.StreamWriter* 的方法), 1101
writelines() (*asyncio.WriteTransport* 的方法), 1150
writelines() (*codecs.StreamWriter* 的方法), 201
writelines() (*io.IOBase* 的方法), 738
writepy() (*zipfile.PyZipFile* 的方法), 598
Writer (*io* 中的類F), 747
writer() (於 *csv* 模組中), 620
writerow() (*csv.csvwriter* 的方法), 625
writerows() (*csv.csvwriter* 的方法), 625
writestr() (*zipfile.ZipFile* 的方法), 596
WriteTransport (*asyncio* 中的類F), 1147
writev() (於 *os* 模組中), 689
writexml() (*xml.dom.minidom.Node* 的方法), 1395
WrongDocumentErr, 1393
wsgi_file_wrapper (*wsgiref.handlers.BaseHandler* 的屬性), 1433
wsgi_multiprocess (*wsgiref.handlers.BaseHandler* 的屬性), 1432
wsgi_multithread (*wsgiref.handlers.BaseHandler* 的屬性), 1431
wsgi_run_once (*wsgiref.handlers.BaseHandler* 的屬性), 1432
WSGIApplication (於 *wsgiref.types* 模組中), 1433
WSGIEnvironment (於 *wsgiref.types* 模組中), 1433
wsgiref
    module, 1426
wsgiref.handlers
    module, 1430
wsgiref.headers
    module, 1428
wsgiref.simple_server
    module, 1428
wsgiref.types
    module, 1433
wsgiref.util
    module, 1426
wsgiref.validate
    module, 1429
WSGIRequestHandler (*wsgiref.simple_server* 中的類F), 1429
WSGIServer (*wsgiref.simple_server* 中的類F), 1429
wShowWindow (*subprocess.STARTUPINFO* 的屬性), 1055
WSTOPPED (於 *os* 模組中), 727
WSTOPSIG() (於 *os* 模組中), 729
wstring_at() (於 *ctypes* 模組中), 845
WTERMSIG() (於 *os* 模組中), 729
WUNTRACED (於 *os* 模組中), 727

WWW, 1423, 1452, 1462
    server（伺服器）, 1511

X

-x
    compileall 命令列選項, 2148
X (於 *re* 模組中), 148
X509 certificate（X509 憑證）, 1227
X_OK (於 *os* 模組中), 691
xatom() (*imaplib.IMAP4* 的方法), 1490
XATTR_CREATE (於 *os* 模組中), 715
XATTR_REPLACE (於 *os* 模組中), 716
XATTR_SIZE_MAX (於 *os* 模組中), 715
xcor() (於 *turtle* 模組中), 1640
xdrlib
    module, 2223
XHTML, 1360
XHTML_NAMESPACE (於 *xml.dom* 模組中), 1385
xml
    module, 1364
XML() (於 *xml.etree.ElementTree* 模組中), 1375
XML_ERROR_ABORTED (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_AMPLIFICATION_LIMIT_BREACH (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_ASYNC_ENTITY (於 *xml.parsers.expat.errors* 模組中), 1418
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (於 *xml.parsers.expat.errors* 模組中), 1418
XML_ERROR_BAD_CHAR_REF (於 *xml.parsers.expat.errors* 模組中), 1418
XML_ERROR_BINARY_ENTITY_REF (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_DUPLICATE_ATTRIBUTE (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_ENTITY_DECLARED_IN_PE (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_EXTERNAL_ENTITY_HANDLING (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_FEATURE_REQUIRES_XML_DTD (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_FINISHED (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_INCOMPLETE_PE (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_INCORRECT_ENCODING (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_INVALID_ARGUMENT (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_INVALID_TOKEN (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_MISPLACED_XML_PI (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_NO_BUFFER (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_NO_ELEMENTS (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_NO_MEMORY (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_NOT_STANDALONE (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_NOT_STARTED (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_NOT_SUSPENDED (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_PARAM_ENTITY_REF (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_PARTIAL_CHAR (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_PUBLICID (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_RECURSIVE_ENTITY_REF (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_RESERVED_NAMESPACE_URI (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_RESERVED_PREFIX_XML (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_RESERVED_PREFIX_XMLNS (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_SUSPEND_PE (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_SUSPENDED (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_SYNTAX (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_TAG_MISMATCH (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_TEXT_DECL (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_UNBOUND_PREFIX (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_UNCLOSED_CDATA_SECTION (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_UNCLOSED_TOKEN (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_UNDECLARING_PREFIX (於 *xml.parsers.expat.errors* 模組中), 1420
XML_ERROR_UNDEFINED_ENTITY (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_UNEXPECTED_STATE (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_UNKNOWN_ENCODING (於 *xml.parsers.expat.errors* 模組中), 1419
XML_ERROR_XML_DECL (於 *xml.parsers.expat.errors* 模組中), 1420
XML_NAMESPACE (於 *xml.dom* 模組中), 1385
xmlcharrefreplace
    error handler's name（錯誤處理器名稱）, 197
xmlcharrefreplace_errors() (於 *codecs* 模組中), 199

XmlDeclHandler() (*xml.parsers.expat.xmlparser* 的方法), 1415
xml.dom
    module, 1384
xml.dom.minidom
    module, 1394
xml.dom.pulldom
    module, 1398
xml.etree.ElementInclude
    module, 1376
xml.etree.ElementTree
    module, 1365
XMLFilterBase (*xml.sax.saxutils* 中的類🄵), 1407
XMLGenerator (*xml.sax.saxutils* 中的類🄵), 1407
XMLID() (於 *xml.etree.ElementTree* 模組中), 1375
XMLNS_NAMESPACE (於 *xml.dom* 模組中), 1385
XMLParser (*xml.etree.ElementTree* 中的類🄵), 1382
xml.parsers.expat
    module, 1411
xml.parsers.expat.errors
    module, 1418
xml.parsers.expat.model
    module, 1418
XMLParserType (於 *xml.parsers.expat* 模組中), 1411
XMLPullParser (*xml.etree.ElementTree* 中的類🄵), 1383
XMLReader (*xml.sax.xmlreader* 中的類🄵), 1407
xmlrpc
    module, 1530
xmlrpc.client
    module, 1530
xmlrpc.server
    module, 1537
xml.sax
    module, 1400
xml.sax.handler
    module, 1401
xml.sax.saxutils
    module, 1406
xml.sax.xmlreader
    module, 1407
xor() (於 *operator* 模組中), 437
xview() (*tkinter.ttk.Treeview* 的方法), 1612

Y

ycor() (於 *turtle* 模組中), 1641
year
    calendar 命令列選項, 263
year (*datetime.date* 的屬性), 221
year (*datetime.datetime* 的屬性), 228
Year 2038 (2038 年問題), 748
yeardatescalendar() (*calendar.Calendar* 的方法), 257
yeardays2calendar() (*calendar.Calendar* 的方法), 257
yeardayscalendar() (*calendar.Calendar* 的方法), 257
YES (於 *tkinter.messagebox* 模組中), 1597
YESEXPR (於 *locale* 模組中), 1572
YESNO (於 *tkinter.messagebox* 模組中), 1597
YESNOCANCEL (於 *tkinter.messagebox* 模組中), 1597
Yield (*ast* 中的類🄵), 2122
YIELD_VALUE (*opcode*), 2160
YieldFrom (*ast* 中的類🄵), 2122
yiq_to_rgb() (於 *colorsys* 模組中), 1560
yview() (*tkinter.ttk.Treeview* 的方法), 1612

Z

z
    於字串格式化, 127
z85decode() (於 *base64* 模組中), 1355
z85encode() (於 *base64* 模組中), 1355
Zen of Python (Python 之🄵) , **2243**
ZeroDivisionError, 116
zfill() (*bytearray* 的方法), 80
zfill() (*bytes* 的方法), 80
zfill() (*str* 的方法), 63
zip()
    built-in function, 31
ZIP_BZIP2 (於 *zipfile* 模組中), 592
ZIP_DEFLATED (於 *zipfile* 模組中), 592
zip_longest() (於 *itertools* 模組中), 419
ZIP_LZMA (於 *zipfile* 模組中), 592
ZIP_STORED (於 *zipfile* 模組中), 592
ZIP_ZSTANDARD (於 *zipfile* 模組中), 592
zipapp
    module, 1914
zipapp 命令列選項
    -c, 1915
    --compress, 1915
    -h, 1915
    --help, 1915
    --info, 1915
    -m, 1914
    --main, 1914
    -o, 1914
    --output, 1914
    -p, 1914
    --python, 1914
zipfile
    module, 591
ZipFile (*zipfile* 中的類🄵), 593
zipfile 命令列選項
    -c, 601
    --create, 601
    -e, 601
    --extract, 601
    -l, 601
    --list, 601
    --metadata-encoding, 601
    -t, 601
    --test, 601
zipimport
    module, 2051
zipimporter (*zipimport* 中的類🄵), 2052
ZipImportError, 2051

ZipInfo (*zipfile* 中的類🄵), 591
zlib
    module, 574
ZLIB_RUNTIME_VERSION (於 *zlib* 模組中), 577
ZLIB_VERSION (於 *zlib* 模組中), 577
ZLIBNG_VERSION (於 *zlib* 模組中), 577
zoneinfo
    module, 251
ZoneInfo (*zoneinfo* 中的類🄵), 253
ZoneInfoNotFoundError, 255
zscore() (*statistics.NormalDist* 的方法), 405
zstd_version_info (於 *compression.zstd* 模組中), 573
ZstdCompressor (*compression.zstd* 中的類🄵), 565
ZstdDecompressor (*compression.zstd* 中的類🄵), 566
ZstdDict (*compression.zstd* 中的類🄵), 568
ZstdError, 564
ZstdFile (*compression.zstd* 中的類🄵), 564